CIS 544 – Advanced Software Design and Development

Project Management System

Daniel Kennedy

Contents

[1. Requirements 4](#_Toc387594608)

[1.1. Use Case Diagram 4](#_Toc387594609)

[1.2. Use Case Descriptions 5](#_Toc387594610)

[1.2.1. Login 5](#_Toc387594611)

[1.2.2. List Project Data By Programmer Name 5](#_Toc387594612)

[1.2.3. List Project Data by Primary Requirement 6](#_Toc387594613)

[1.2.4. Display Information on a Selected Program Unit 6](#_Toc387594614)

[1.2.5. Display Statistics for all programming units 7](#_Toc387594615)

[1.2.6. Add a New Programming Unit 7](#_Toc387594616)

[1.2.7. Edit a Programming Unit 8](#_Toc387594617)

[1.2.8. Delete a Programming Unit 8](#_Toc387594618)

[1.2.9. Quit 9](#_Toc387594619)

[1.3. Domain Analysis Diagram 11](#_Toc387594620)

[2. Architecture 12](#_Toc387594621)

[2.1. Component Diagram 12](#_Toc387594622)

[2.1.1. Component Diagram Interfaces 13](#_Toc387594623)

[3. Mid-Level Design 17](#_Toc387594624)

[3.1. System Class Diagram 17](#_Toc387594625)

[4. Low-Level Design 17](#_Toc387594626)

[4.1. Operation Specifications 17](#_Toc387594627)

[5. Translation to Code 20](#_Toc387594628)

[6. Screenshots of Use 21](#_Toc387594629)

[6.1. Screenshots of a Manager’s Use 21](#_Toc387594630)

[6.1.1. Login Screen 21](#_Toc387594631)

[6.1.2. Main Menu 22](#_Toc387594632)

[6.1.3. Adding a new Unit 22](#_Toc387594633)

[6.1.4. Editing a Unit 23](#_Toc387594634)

[6.1.5. Deleting a Unit 24](#_Toc387594635)

[6.1.6. Quitting 24](#_Toc387594636)

[6.2. Screenshots of a User’s Use 24](#_Toc387594637)

[6.2.1. Main Menu 24](#_Toc387594638)

[6.2.2. Listing by Programmer Name 25](#_Toc387594639)

[6.2.3 Listing by Primary Requirement 26](#_Toc387594640)

[6.2.4 Display a Single Programming Unit 27](#_Toc387594641)

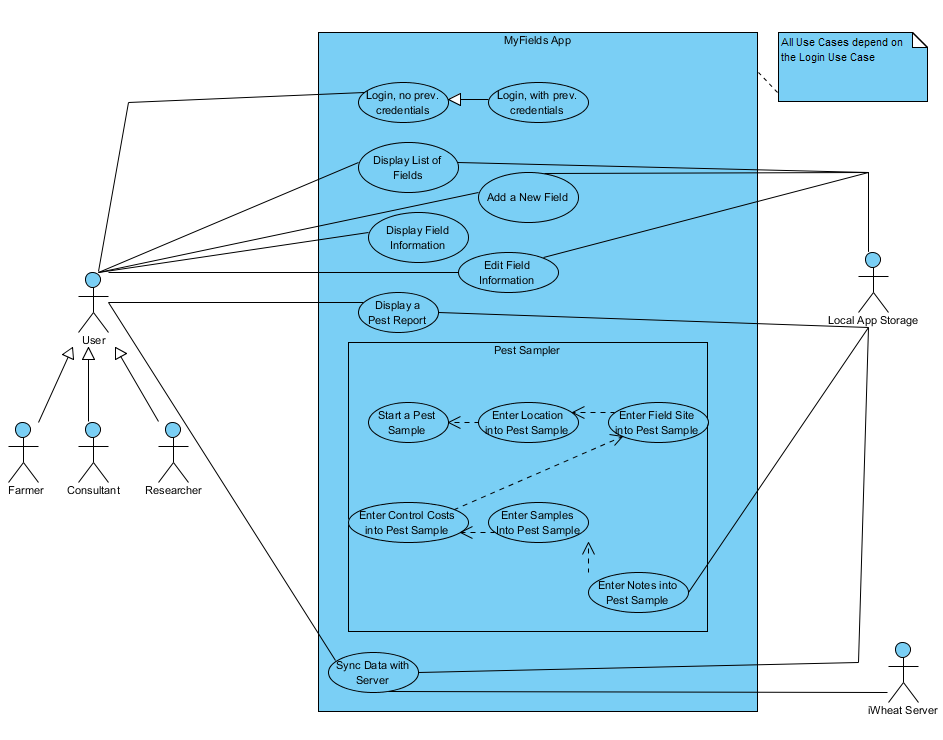
[6.2.5 Display Statistics 27](#_Toc387594642)

[6.2.6 Quitting 27](#_Toc387594643)

[7. Conclusion 27](#_Toc387594644)

# Requirements

## Use Case Diagram



## Use Case Descriptions

### 1.2.1. Login, no previous credentials

**Actors**: User

**Stakeholders and Needs**:

User – Login to system to interact.

**Preconditions**: User has started system.

**Postconditions**: Fields List is displayed.

**Trigger**: User starts system.

**Basic Flow**:

1. System displays form with inputs for username and password, and login button
2. User inputs username and password
3. User clicks “Login” button
4. System authenticates with iWheat.org to retrieve user’s fields and associated info
5. System stores retrieved info and displays it in Fields List page

**Extensions**:

4a. If user credentials are invalid, return to step 1 and display “Invalid username/password.”

### 1.2.2. Login, with previous credentials

**Actors**: User

**Stakeholders and Needs**:

User – Login to system to interact.

**Preconditions**: User has started system, and has logged in to system previously.

**Postconditions**: Fields List is displayed.

**Trigger**: User starts system.

**Basic Flow**:

1. System retrieves credentials from secure local storage
2. System authenticates with iWheat.org to retrieve user’s fields and associated info
3. System checks retrieved info against locally stored info
4. System updates local storage to the most recent info and displays Fields List page

**Extensions**:

2a. If user credentials are invalid, go to step 1 of use case 1.2.1.

4a. If information does not match, fields have been updated while app was closed, via the website. Website information should take precedence.

### 1.2.3. Display List of Fields

**Actors**: User

**Stakeholders and Needs**:

User – View list of all fields, scroll through list

**Preconditions**: User is logged in to the system

**Postconditions**: User sees list of fields displayed

**Trigger**: User clicks “Login” with valid credentials OR valid credentials have been detected from a previous run

**Basic Flow**:

1. System uses user credentials to retrieve user’s field information from the website.
2. System parses retrieved information into Fields objects.
3. System checks local storage for Fields objects.
4. System builds and displays a form with each of those Fields objects to the user.

**Extensions**:

3a. If local fields objects are detected, and they do not match, the more recent object should be written into local storage.

### 1.2.4. Add a New Field

**Actors**: User

**Stakeholders and Needs**:

User – Add a new field to the list of user’s fields

**Preconditions**: User is logged in to the system

**Postconditions**: User has a new field displayed in their list of fields.

**Trigger**: User clicks on the “Add Field…” button on the Fields List page, or “Add a Field” from the Pest Sampler “Select a Field” page.

**Basic Flow**:

1. System builds and displays a form asking for each piece of information necessary to build a new field, with buttons for “Save” and “Cancel”
2. User inputs required information, and any optional information.
3. User clicks “Save” or “Cancel.”
   1. If User clicks Save, a new Field object is created from the entered information
   2. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and Fields List is displayed
      2. If “No” is pressed, User returns to previous form.
4. System builds and displays the updated form with each field, including the new, to the user.

**Extensions**:

1a. ABSOLUTE necessary information is currently: Field Name, Latitude, Longitude

1b. Additional optional information: Size (acres), Soil Type, Tillage System, Irrigation Type

3a. Field Name, Latitude, Longitude, and Size should be validated. Other values are dropdowns.

### 1.2.5. Display Field Information

**Actors**: User

**Stakeholders and Needs**:

User – View information related to a specific field.

**Preconditions**: User is logged in to the system

**Postconditions**: User sees all information related to a specific field

**Trigger**: User clicks on a specific field in the Fields List OR clicks a link to that field from a page related to that field.

**Basic Flow**:

1. System retrieves the Field object associated with the named field.
2. System builds and displays a form containing all information associated with that field, including previous pest samples.

**Extensions**:

2a. Information to Display: Field Name, Latitude, Longitude, Size (acres), Soil Type, Tillage System, Irrigation Type

### 1.2.6. Edit a Field

**Actors**: User

**Stakeholders and Needs**:

User – Edit information associated with a field.

**Preconditions**: User is logged in to the system, and is on the desired field’s information page

**Postconditions**: System updates field information.

**Trigger**: User clicks on the “Edit Field” Button on the Field Information page.

**Basic Flow**:

1. System modifies the Field Information form to have editable fields, and a “Save” and “Cancel” button.
2. User clicks on the desired area to edit, and inputs the desired information.
3. User clicks “Save” or “Cancel.”
   1. If User clicks Save, that field object is updated with the new information.
   2. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and Fields Information page is displayed.
      2. If “No” is pressed, User returns to previous form.
4. System returns to the Field Information form, which is updated if necessary.

**Extensions**:

1a. Necessary information is currently: Field Name, Latitude, Longitude, Size (acres), Soil Type, Tillage System, Irrigation Type

3a. Field Name, Latitude, Longitude, and Size should be validated. Other values are dropdowns.

### 1.2.7. Start a Pest Sample

**Actors**: User

**Stakeholders and Needs**:

User – Create a pest sample for a field.

**Preconditions**: User is logged in to the system

**Postconditions**: User is taken to the Sampling Method page of the Pest Sampler wizard

**Trigger**: User clicks on Pest Sampler page in app.

**Basic Flow**:

1. System builds and displays a form asking for the Sampling Method (dropdown) input, with buttons to “Continue” or “Cancel”
2. User clicks the dropdown and clicks one of the values.
3. User clicks either “Continue” or “Cancel”
   1. If Continue is clicked, the system will create a new pest sample object to modify with further information.
   2. The system will then display the “Location” page of the Pest Sampler.
   3. If Cancel is clicked, the system will return the user to the Fields List page.

**Extensions**:

1a. Dropdown values: Glance N Go (Greenbug), Wheat Stem Sawfly

3a. At this point, the system should validate that the user has selected a value in the dropdown; if not, return to step 2 and display a message stating “Select a sampling method!”.

### 1.2.8. Enter a location into the Pest Sampler

**Actors**: User

**Stakeholders and Needs**:

User – Create a pest sample for a field.

**Preconditions**: User is logged in to the system, has completed the “Sampling Method” page, and clicked Continue.

**Postconditions**: User is taken to the Site page of the Pest Sampler wizard

**Trigger**: User clicks on Continue on the “Sampling Method” page.

**Basic Flow**:

1. System builds and displays a form showing the detected device location on a map.
2. User may wish to move locations; should be able to if so desired.
3. User clicks either “Continue,” “Cancel,” or “Back”
   1. If Continue is clicked, the system will add the centerpoint location on the map to the created Pest Sample.
   2. The system will then display the “Sample Site” page to the user.
   3. If “Back is clicked, the system will display the “Sampling Method” page to the user.
   4. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and Fields List page is displayed.
      2. If “No” is pressed, User returns to previous form.

**Extensions**:

3a. At this point, the system should validate that the user has centered the map over a valid location; if not, return to step 2 and display a message “Select a valid location!”.

### 1.2.9. Enter a Field Site into the Pest Sampler

**Actors**: User

**Stakeholders and Needs**:

User – Create a pest sample for a field.

**Preconditions**: User is logged in to the system, has completed the “Sampling Method” page and “Location” page, and clicked Continue.

**Postconditions**: User is taken to the Costs page of the Pest Sampler wizard

**Trigger**: User clicks on Continue on the “Location” page.

**Basic Flow**:

1. System builds and displays a form showing a dropdown with a list of the user’s fields, or the option to add a new field, with a “Continue,” “Back,” and “Cancel” buttons.
2. User will input either a dropdown value or “Add a new field” checkbox
   1. If User selects to add a new field, the system will display a box showing the previously selected location and asking for the field name, with a “Create” button.
   2. The user will enter the field name, then select “Create”
3. User clicks either “Continue,” “Cancel,” or “Back”
   1. If Continue is clicked, the system will do one of the two following:
      1. Create a new field based on the inputted name and location.
         1. Link the current pest sample to the new field.
      2. Link the current pest sample to the selected field.
   2. The system will then display the “Costs” page to the user.
   3. If “Back is clicked, the system will display the “Location” page to the user.
   4. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and Fields List page is displayed.
      2. If “No” is pressed, User returns to previous form.

**Extensions**:

3a. At this point, the system should validate that either a dropdown value has been selected or a new field entered; if not, return to step 2 and display a message stating “Select or create a field!”.

### 1.2.10. Enter Control Costs into the Pest Sampler.

**Actors**: User

**Stakeholders and Needs**:

User – Create a pest sample for a field.

**Preconditions**: User is logged in to the system, has completed the “Sampling Method” page, “Location” page, and “Field Site” page, and clicked Continue.

**Postconditions**: User is taken to the “Samples” page of the Pest Sampler wizard

**Trigger**: User clicks on Continue on the “Field Site” page.

**Basic Flow**:

1. System builds and displays a form showing two dropdown with headers for Control Cost and Crop Value, with a “Continue,” “Back,” and “Cancel” buttons.
2. User will click the Control Cost dropdown, and enter a value.
3. User will click the Crop Value dropdown, and enter a value.
4. User clicks either “Continue,” “Cancel,” or “Back”
   1. If Continue is clicked, the system will add the inputted control and crop value costs to the pest sample.
   2. The system will then display the “Samples” page to the user.
   3. If “Back is clicked, the system will display the “Field Site” page to the user.
   4. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and Fields List page is displayed.
      2. If “No” is pressed, User returns to previous form.

**Extensions**:

1a. Valid values for Control Cost: 4-12, integer values

1b. Valid values for Crop Value: 2.5 – 7, incrementing by .5

4a. At this point, the system should validate that both dropdown values have been entered; if not, return to step 2 and display a message stating “Select both control cost and crop value!”.

### 1.2.11. Enter Samples into the Pest Sampler.

**Actors**: User

**Stakeholders and Needs**:

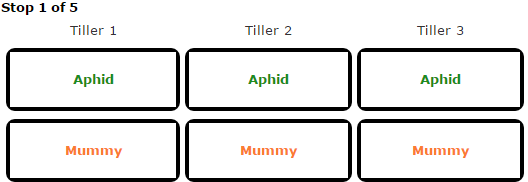
User – Create a pest sample for a field.

**Preconditions**: User is logged in to the system, has completed the “Sampling Method” page, “Location” page, “Field Site” page, and “Costs” page, and clicked Continue.

**Postconditions**: User is taken to the “Notes” page of the Pest Sampler wizard

**Trigger**: User clicks on Continue on the “Costs” page.

**Basic Flow**:

1. System builds and displays a form showing a table with checkboxes as the following:
   1.  with “Continue,” “Back,” and “Cancel” buttons.
2. User will select checkboxes on the table.
3. User clicks either “Continue,” “Cancel,” or “Back”
   1. If Continue is clicked, the system will add to the total number of detected Aphids and Mummys based on the checkboxes selected to the current sample object.
   2. The system will then display the next “Sample” page to the user, or Notes if 5 samples have been collected.
   3. If “Back is clicked, the system will display the “Costs” page to the user if it is the first sample, or the previous “Sample” page otherwise.
   4. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and Fields List page is displayed.
      2. If “No” is pressed, User returns to previous form.

**Extensions**:

4a. At this point, the system should validate that the user has selected at least one checkbox; if not, display a message stating “Select at least one value!”.

### 1.2.12. Enter Notes into the Pest Sampler

**Actors**: User

**Stakeholders and Needs**:

User – Create a pest sample for a field.

**Preconditions**: User is logged in to the system, has completed the “Sampling Method” page, “Location” page, “Field Site” page, “Costs” page, and “Samples” pages, and clicked Continue.

**Postconditions**: User is taken to the “Pest Report” page of the Pest Sampler wizard

**Trigger**: User clicks on Continue on the last “Samples” page.

**Basic Flow**:

1. System builds and displays a form showing text entry boxes for Notes and Other Pests Observed with “Continue,” “Back,” and “Cancel” buttons.
2. User will either enter text into the boxes or not.
3. User clicks either “Continue,” “Cancel,” or “Back”
   1. If Continue is clicked, the system will add the desired notes and other observed pests to the current sample.
   2. The system will then display the “Pest Report” page to the user.
   3. If “Back is clicked, the system will display the last “Samples” page to the user.
   4. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and Fields List page is displayed.
      2. If “No” is pressed, User returns to previous form.

**Extensions**:

3a. It is not required for the user to enter any text, so text should not be validated for notes.

### 1.2.13. Display a Pest Report

**Actors**: User

**Stakeholders and Needs**:

User – Review a pest sample associated with a field

**Preconditions**: User is logged in to the system, has completed all pages within the Pest Sampler and clicked “Continue,” OR clicks on a pest sample displayed on a Field Information page.

**Postconditions**: User is taken to the “Pest Report” page.

**Trigger**: User clicks on Continue on the “Notes” page of the Sampler, or clicks on a pest sample displayed on a Field Information page.

**Basic Flow**:

1. System builds and displays a form showing all information associated with a pest sample, with a button to “Edit” the sample.
2. If the user clicks “Edit,” all information becomes editable boxes the user may click and change, and a “Save” and “Cancel” button shall be displayed at the bottom of the page.
3. If the user clicks “Save”, all information shall be compared to the last known information for the sample, and if a mismatch occurs then the new information shall overwrite the old.
   1. If User clicks Cancel, “Are you sure?” is displayed in a message box with “Yes” or “No” options.
      1. If “Yes” is pressed, information is discarded and the Pest Report page is displayed.
      2. If “No” is pressed, User returns to previous form.

**Extensions**:

1a. All information includes: Location (map), Field Site, Sampling Method, Control Cost, Crop Value, Threshold for Treatment, Recommendation, Stop Data (stop #, Greenbug count, Mummy count), Notes, and Other Pests Observed.

### 1.2.14. Sync Data with Server

**Actors**: User

**Stakeholders and Needs**:

User – Store all local information with the iWheat server.

**Preconditions**: User is logged in to the system

**Postconditions**: All local information to the app is synchronized with the iWheat server’s information

**Trigger**: User selects the menu and clicks “Sync,” OR performed periodically.

**Basic Flow**:

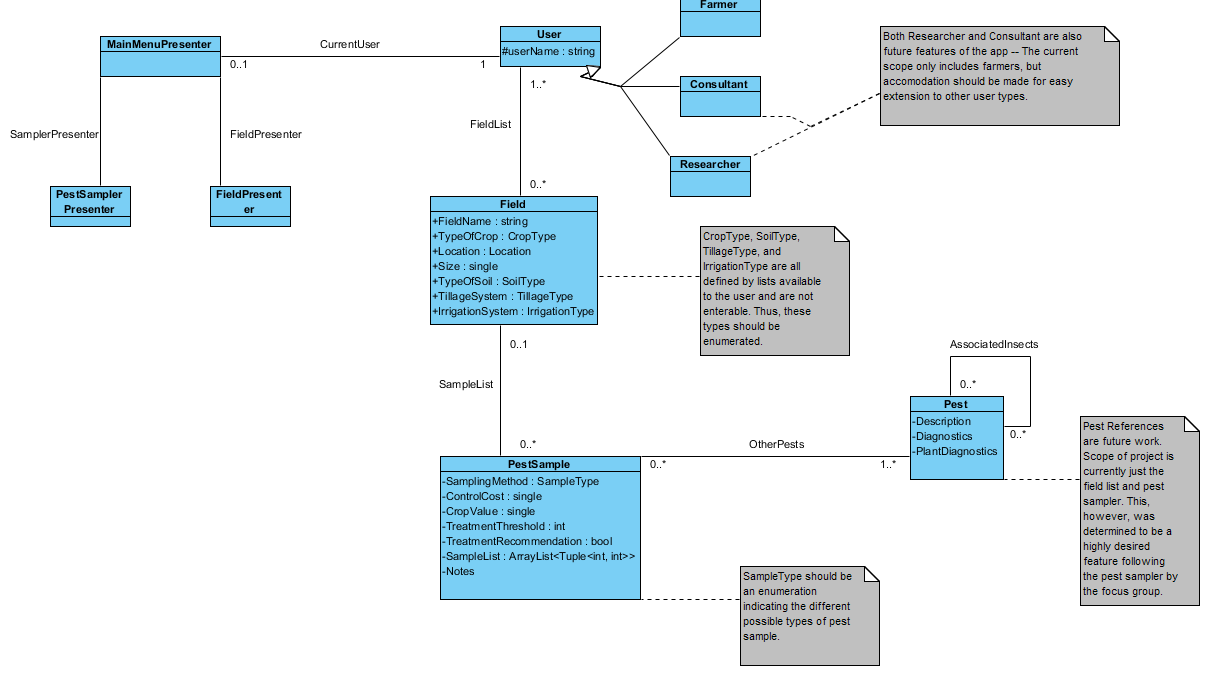
1. System retrieves all field information for the user from the iWheat server, and constructs Fields objects with linked Pest Samples.
2. System compares the created objects; if a mismatch is detected, that field and associated pest samples are put in to a list for synchronization with the server.
   1. If a field is not in the list provided by the server, it will also be added to the sync list.
3. Once all fields have been compared, the system will serialize each field marked for sync to the iWheat server.
4. The server will store that data for the user for later usage and retrieval.

**Extensions**:

2a. If a field is provided by the server that is not detected by the app, that field should be added to the app’s local list.

# Architecture

## Domain Analysis

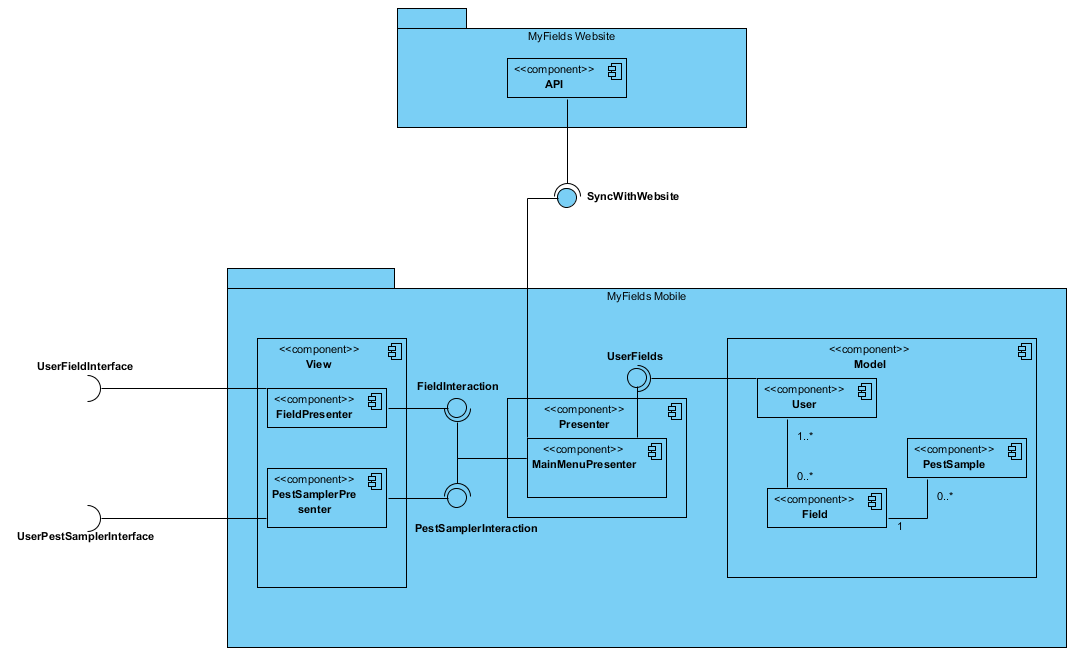


## Domain Analysis Description

An analysis of the domain reveals three main data objects: a user, a field, and a pest sample. A user may own one or many fields, and a field may have one or many pest samples. These data types serve to make up the distinct data set that the app will be interacting with. With such a clear data model, and the need for a modular system, a Model-View-Presenter makes sense in this architecture. This app is also intended to work similarly to a website; this is another reason the MVP architecture makes sense. Most websites make use of a Model-View-Controller approach; MVP is very similar, except it decouples the view from the model.

This app must be designed for ease of maintainability and ease of extension, as this is a project intended to stretch out many years into the future. To that extent, it makes sense for each area of the app to have a separate handler for each function of the app. FieldPresenter and PestSampler presenter handle each of the current intended functions of the app – to be able to add, view, and edit field information, and to create pest samples to link to a field. MainMenuPresenter is intended to act as an interface between the functional handlers and the data on the back end (e.g. the fields and pest samples). By this design, the system acts as a Model-View-Presenter system. The functional handlers act as the view, the MainMenuPresenter acts as the main controller, and the field, user, and pest sample objects act as the data. This design allows for ease of extension of the system; to add a new function, the only necessary change is to add a new Presenter on the front end, and any additional data on the back end. The MainMenuPresenter acts as an interface in between the two systems, gathering and collecting the necessary data from the Model and passing it to the View.

## Component Diagram



## Component Diagram Description

This component diagram gives a visual representation of the responsibilities of each piece of the MyFields Mobile website. Each section has been lumped into components in the Model-View-Presenter design schema to aid in visualizing the breakdown of responsibilities in this design. As noted, the FieldPresenter and PestSamplerPresenter both act as part of the view, interfacing with the user. The main presenter component is the MainMenuPresenter, which acts as both an interface between the Model and the View, and as an interface to synchronize data with the website. The Model consists of the necessary user information and potentially preferences, as well as that user’s field information and those fields’ associated pest samplings.

### The View

The View component acts as the user interface for the system. The current design has two main functions: to display, manage, edit, and add new fields; and to create pest samples associated with those fields. Thus, to make for a modular system, the View component needs two different interfaces. These interfaces are provided by the FieldPresenter and PestSamplerPresenter components. By breaking the view into components for each function of the app, it makes it easy to extend the app in the future; simply add a new user interface in the view, add any necessary components to the Model data, and link them with the Presenter.

#### Interfaces

The View component has four interfaces; 2 per inner component. The first interface is the UserFieldInterface for field information. This interface should provide a means for the user to add, edit, and view information about each of their fields individually, or a list of the fields overall. The second interface is from the FieldPresenter to the MainMenuPresenter; called FieldInteraction. This interface should provide all necessary information about one particular field, or a list of all fields in general, for the FieldPresenter to display to the user.

The third interface is the UserPestSamplerInterface for creating pest samples. This interface should provide a means for the user to create new pest samples to link to the fields. The fourth interface is the PestSamplerInteraction interface between the PestSamplerPresenter and the MainMenuPresenter. This interface should provide a field index to link to the Pest Sample, and should return a created pest sample for the Presenter to add to the Model, linked to the Field provided.

### The Presenter

The Presenter component acts as the interface between the Model and the View components. By decoupling these, this makes the overall system more modular, and requires less modification when extending the app into the future. The Presenter component will also act as an interface between the MyFields website and the Model storage, synchronizing the data stored between the two once per hour. This functionality adds substance to the Presenter class, so it is more than just a go-between for the View and Model.

#### Interfaces

The Presenter has four interfaces, and is the most highly coupled component in the system. The first interface must be from the MainMenuPresenter component to the User component in the Model. See [section 2.4.3.1](#_Interfaces) for information about this interface. The second interface must be from the MainMenuPresenter to the API of the MyFields website. This interface will synchronize the data contained within the Model to the information contained by the website about that user. The third and fourth interfaces are very similar. The third is to the FieldPresenter component of the system from MainMenuPresenter; see section 3.2.1.1 for information about this interface. The fourth is from MainMenuPresenter to the PestSamplerPresenter in the View component. See [section 2.4.1.1](#_Interfaces_1) for information about this interface.

### The Model

Currently, there are only three types of data that need to be stored by the app. The system must have a user, that owns a set of fields (a user may have no fields), with a potential set of preferences for the app. The app must also store all information about that user’s fields. This should only be current information about the fields, as the website will store all previous field information. The last piece of data stored in the model is the Pest Samples. These objects are associated with a particular field, and contain information such as the cost of the field’s crop, the cost to treat for that sample’s pest, and a recommendation of whether to treat the field or not. A new pest sample will be built each time the Pest Sampler is used.

#### Interfaces

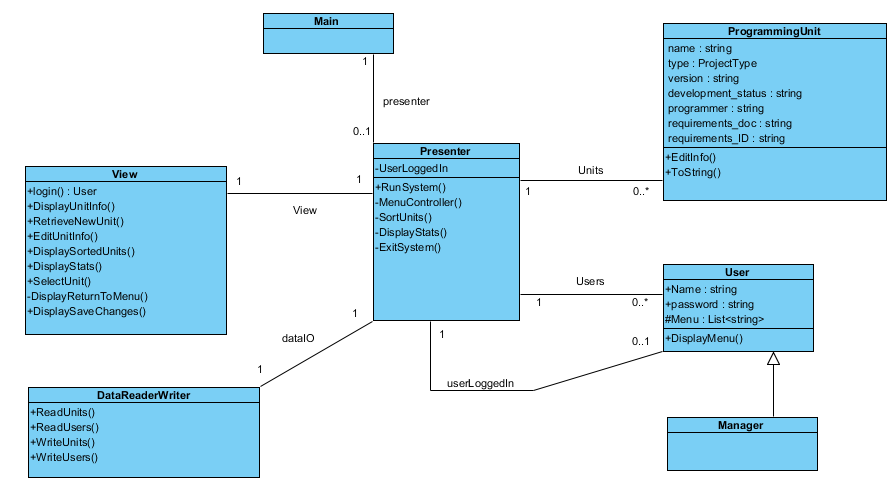
The Model only has one interface. This interface is provided by the Model component (specifically the user object) to the Presenter component; specifically the MainMenuPresenter. This interface must provide the MainMenuPresenter with the Model’s list of fields, their associated Pest Samples (which should be packaged in the field) and any user preferences related to the request by the MainMenuPresenter.

### Component Diagram Interfaces

# Mid-Level Design

## System Class Diagram

Operation signatures are hidden in this section for clarity of the diagram. See section 4.1 for specifications.



The Presenter object acts as a centralized controller for this system, while the View object acts as a user interface, and the DataReaderWriter acts as an interface for reading to and writing from local storage for the project. The ProgrammingUnit and User objects act as the Models for this system.

## Low-Level Design

## Operation Specifications

1. View Object
   1. Login() : User
      1. Purpose: To provide an interface for a user to login to the system or create an account.
   2. DisplayUnitInfo(ProgrammingUnit p)
      1. Purpose: To provide a means of displaying the information about a ProgrammingUnit
   3. RetrieveNewUnit(): ProgrammingUnit
      1. Purpose: To retrieve all necessary information about creating a programming unit, then create and return that unit.
   4. EditUnitInfo(ProgrammingUnit p)
      1. Purpose: To provide an interface for selecting pieces of a programming unit to modify
   5. DisplaySortedUnits(List<ProgrammingUnit> units)
      1. Purpose: To display a sorted list of programming units.
   6. DisplayStats(int NumberNotStarted, int NumberDesign, int NumberCoding, int NumberTesting)
      1. Purpose: To display the number of ProgrammingUnits in each cycle of development.
   7. SelectUnit(List<ProgrammingUnit> units): int
      1. Purpose: To provide the user a method of selecting the index of a programming unit to modify or delete.
   8. DisplayReturnToMenu()
      1. Purpose: To provide a centralized piece of code for the end of each display method to call.
   9. DisplaySaveChanges()
      1. Purpose: To provide a centralized piece of code for prompting to save changes (internal or external) with internal error handling.
2. DataReaderWriter Object
   1. ReadUnits(): List<ProgrammingUnit>
      1. Purpose: To read in a group of programming units from a database file.
   2. ReadUsers() : List<User>
      1. Purpose: To read in a group of users from a database file.
   3. WriterUnits(List<ProgrammingUnit> units)
      1. Purpose: To write out the program’s list of programming units to a database file.
   4. WriteUsers(List<User> users)
      1. Purpose: To write out the program’s list of users to a database file.
3. Presenter Object
   1. RunSystem()
      1. Purpose: To provide an entry point for the program, determine if a user needs to log in, display the main menu, and determine whether to continue to selection functionality or exit the system.
   2. MenuController()
      1. Purpose: To determine what to do with a user’s selection from the main PMS menu.
   3. SortUnits(SortValue s)
      1. Purpose: To provide a means of sorting the list of programming units by the specified method. NOTE: SortValue is an enumeration defined within the project for sorting methods; currently either by Programmer Name or Primary Requirement.
   4. DisplayStats()
      1. Purpose: To calculate and pass to the view the statistics of how many programming units are in each stage of the development cycle.
   5. ExitSystem()
      1. Purpose: To determine whether the state of the system needs to be saved, and if so, to save the information before exiting the system.
4. ProgrammingUnit Object
   1. EditInfo(int selectionToEdit, string editText)
      1. Purpose: To provide a centralized input area for editing data fields of a programming unit.
   2. ToString() : string
      1. Purpose: To provide an organized output to the user of the data fields of a programming unit.
5. User Object
   1. DisplayMenu() : int
      1. Purpose: To display the user-specific PMS main menu and return the user’s selection from that menu.

# Translation to Code

For my language of choice, I chose to use C#, since that is the language I have the most experience programming in. I used the Visual Paradigm code generator feature to create a general outline of my project. After that was generated, I started in the Main function of the project and began to trace through the use cases one by one. I implemented each function in the order that they were called by the project; so, for Login as an example, I generated the Presenter object in the Main method, then called the RunSystem() method from there. After that was done, I went to the Presenter object and implemented the necessary code for calling the login() method of the View object. Following the route, I then went to the View object and implemented the login() function.

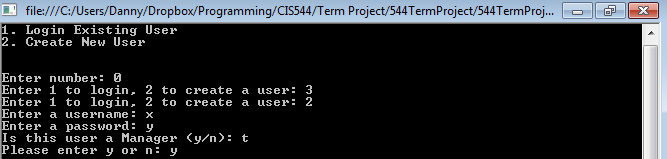
There were certain snippets that had to be done out of order, however. For example, the SelectUnit() function in the View object had to be implemented before I could implement either the use case for editing a programming unit or deleting one, since both of those functions relied on the same method. Similarly, ToString() in the ProgrammingUnit class needed to be done before I could implement DisplayUnitInfo() in the View object, since that function relied on the ToString() functionality in a ProgrammingUnit.

Translation from associations to attributes was a breeze. Everything except the collections of objects was a 1-to-1 or 0..1-to-1 multiplicity, which just meant that the attribute had to be instantiated when that object was created for the former, or that it had to be instantiated at the right moment for the latter. The collections were also easy, as they start out as empty lists and had no restriction on size limit.

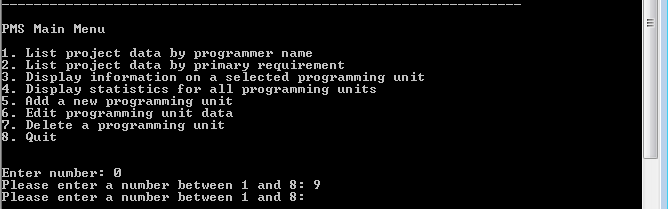
# Screenshots of Use

## Screenshots of a Manager’s Use

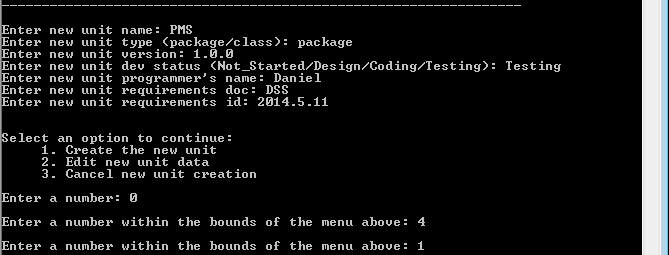
### Login Screen



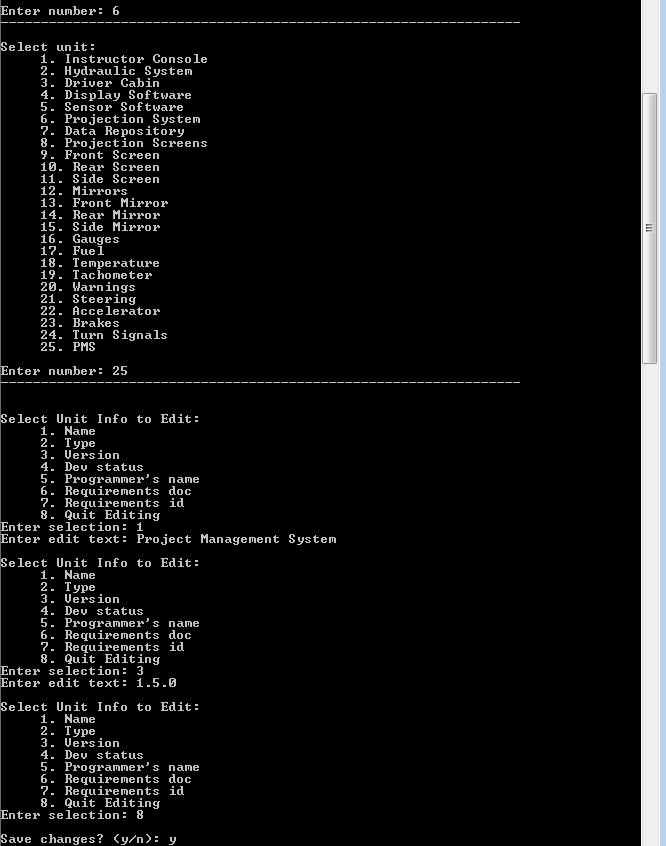
### Main Menu



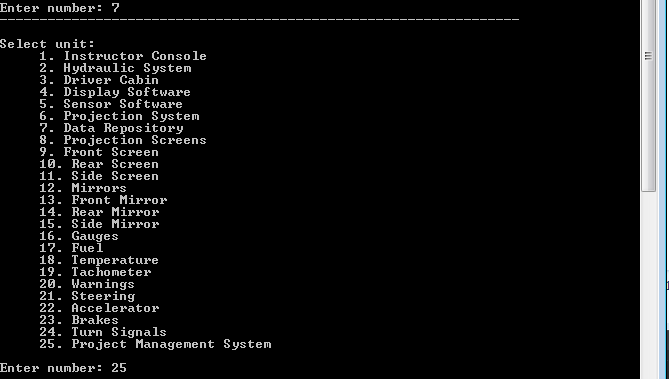
### Adding a new Unit



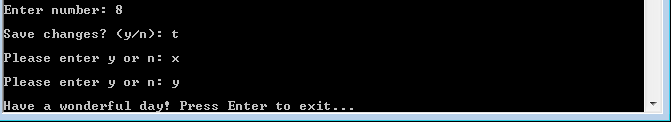
### Editing a Unit



### Deleting a Unit

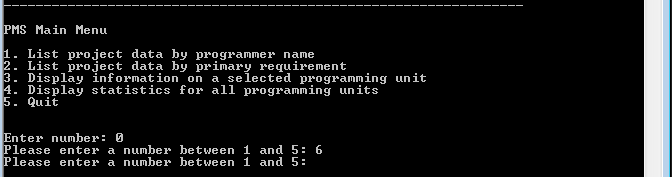


### Quitting

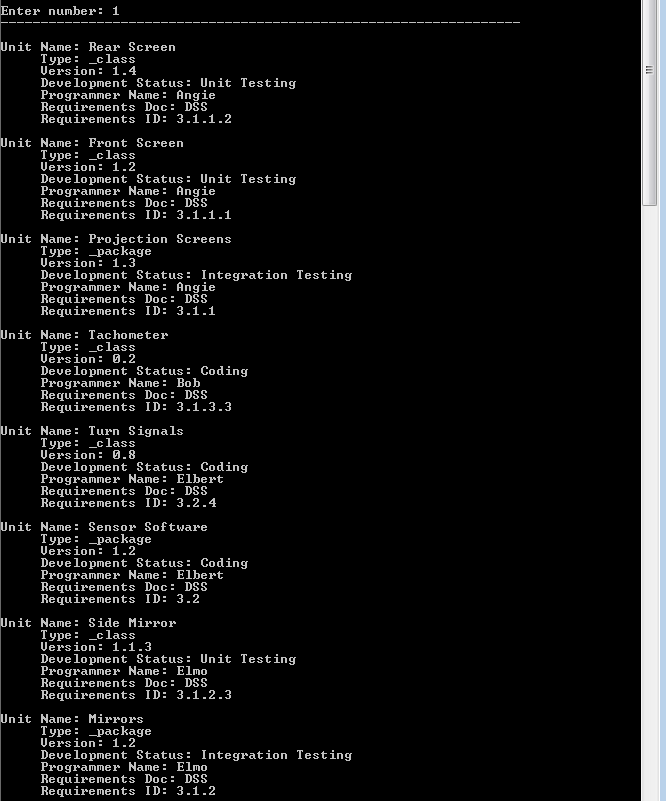


## Screenshots of a User’s Use

### Main Menu



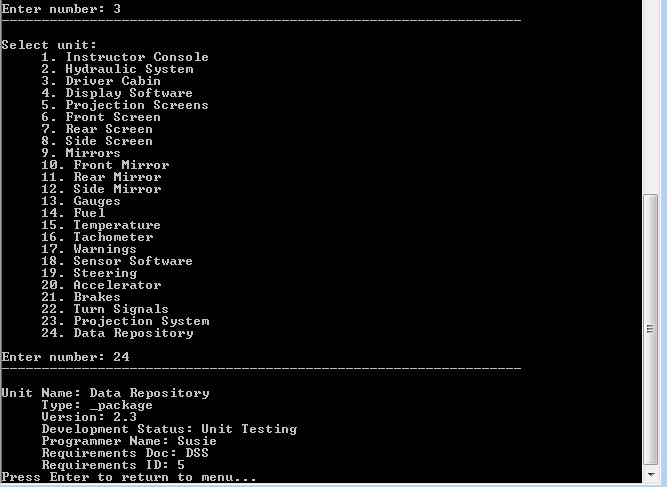
### Listing by Programmer Name



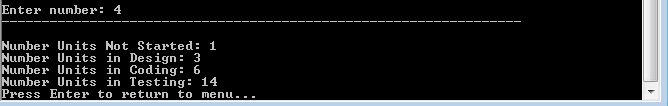
### Listing by Primary Requirement



### Display a Single Programming Unit



### Display Statistics



### Quitting



# Conclusion

All requirements were met in the final program. This was a nice return to coding for me, as the last several classes I have had in the CIS department were more low-level, memory management things. While the benefits of designing a program first through the use of diagrams such as components and class diagrams (of all levels of design) are obvious even in such a small system, it would have been nice to have had the time to design a larger system. In a small-scale project such as this, the benefits are a little bit less, and you don’t need to rely on the diagrams because the design is small enough to keep in your mind. Because of that, I feel that I strayed from the Model-View-Presenter architecture once or twice by putting Presenter logic within the View object, and such things as that. In a larger system, where control may have to be distributed amongst several controllers or components, it should be much harder to go off the design in your head rather than on the diagrams, and it ought to be easier to stick to the architecture model prescribed.

All in all, the benefits of software design are still immediate and obvious. For the first time, I think I actually followed the ideal of spending more time planning than coding or debugging; when I actually reached debugging, due to a reasonable design, there were very few bugs in the code that I had to account for.