1997

Object-oriented design and implementation of INVADERS System

Jun Huang

The University of Montana

Let us know how access to this document benefits you.
Follow this and additional works at: https://scholarworks.umt.edu/etd

Recommended Citation
Huang, Jun, "Object-oriented design and implementation of INVADERS System" (1997). Graduate Student Theses, Dissertations, & Professional Papers. 5544.
https://scholarworks.umt.edu/etd/5544

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.
The University of MONTANA

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

** Please check "Yes" or "No" and provide signature **

Yes, I grant permission
No, I do not grant permission

Author's Signature

Date 11/24/97

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.
Object-oriented Design and Implementation of
INVADERS System

by

Jun Huang

B.S. Shanghai University of Science and Technology, 1988
presented in partial fulfillment of the requirements
for the degree of
Master of Science
The University of Montana
November 1997

Approved by:

Chairperson

Dean, Graduate School

Date
The purpose of INVADERS software system is to build a computerized catalogue of plant names and an enhanced electronic atlas of weed distribution data. The software is designed for ease of use and might be considered to be a computerized companion to "Weeds of the West". The INVADERS System was completed using Booch's[1] object-oriented methodology. The programming language was C++ and the Microsoft Foundation Class Library was used.
# Table of Contents

1. **Introduction**
   - 1.1 Purpose of the INVADERS Software Project ..................................9
   - 1.2 Design Goals .........................................................................................10
   - 1.3 Potential User Group ............................................................................11
   - 1.4 Some Possible Extensions ...............................................................12
   - 1.5 Status of INVADERS Database ...........................................................14

2. **Object Oriented Model**
   - 2.1 Object Oriented Design Definition ..................................................17
   - 2.2 Advantage of Object-Oriented Model ...............................................20
   - 2.3 Notation ..................................................................................................21
     - 2.3.1 Class Diagram Notation ...................................................22
     - 2.3.2 Object Diagram Notation ...............................................24
     - 2.3.3 State Transition Diagram Notation ..................................25
     - 2.3.4 Interaction Diagram Notation ........................................26

3. **System Analysis**
   - 3.1 System Requirements ........................................................................27
   - 3.2 System Analysis ....................................................................................30
     - 3.2.1 Defining the Boundaries of the Problem ..................................30
     - 3.2.2 Database Models ...........................................................................36

4. **System Design**
   - 38
4.1 Class Diagrams .................................................................38
4.2 Object Diagram .............................................................39
4.3 State Transition Diagrams ..............................................41
4.4 Interaction Diagrams of Key Scenarios .........................43
  4.4.1 Query Database by Plant_ID ..................................45
  4.4.2 Display Time Lapse Distribution Map ....................45
  4.4.3 Printing .................................................................47

5. Implementation .................................................................49
  5.1 Understanding ODBC ..................................................49
  5.2 The Role of SQL in ODBC ..........................................51
  5.3 Understanding MFC ......................................................52
  5.4 User Interface ............................................................53
    5.4.1 Basic Frame Window .........................................53
    5.4.2 Query Dialog .....................................................54
  5.5 Query Result Display ................................................57
    5.5.1 Message Box ......................................................57
    5.5.2 Display Table .....................................................58
    5.5.3 Display Plot .......................................................59
    5.5.4 Display Distribution Map ....................................59
    5.5.5 Display Time Lapse Distribution Map ................60
6. Testing

6.1 Testing and Results .................................................................62
    case 1 ..................................................................................62
    case 2 ..................................................................................67
    case 3 ..................................................................................71
    case 4 ..................................................................................76

6.2 Test Cases Evaluation .........................................................77
    case 1 ..................................................................................77
    case 2 ..................................................................................77
    case 3 ..................................................................................77
    case 4 ..................................................................................77
    user interface ........................................................................78

7. Discussion and Conclusion .................................................79

7.1 General Discussion ..............................................................79

7.2 Suggestion of Future Work ..................................................79

7.3 Summary ............................................................................80

Appendix A - Class Specifications ........................................81

Appendix B - Database Fields List ..........................................95

Reference ................................................................................99
List of Figures

Figure 2.1 Class Icon .................................................................22
Figure 2.2 Class Relation – Association ......................................23
Figure 2.3 Class Relation – Use ..................................................23
Figure 2.4 Class Relation – Has ..................................................24
Figure 2.5 Object Icon ...............................................................24
Figure 2.6 State Transition Diagram Icon ....................................25
Figure 3.1 Query Result Message Box ........................................28
Figure 3.2 Query Result – Plot ....................................................29
Figure 3.3 Query Result – Distribution Map ...............................30
Figure 3.4 Relationship Between Frame window and View ............33
Figure 3.5 Relationship of Document-View-Data .......................34
Figure 4.1 INVADERS System Class Diagram ..........................38
Figure 4.2 INVADERS Object Diagram .....................................41
Figure 4.3.1 STD for Class DstrbSet ..........................................42
Figure 4.3.2 STD for Class QueryDialog ....................................42
Figure 4.3.3 STD for Class View ...............................................43
Figure 4.3.4 STD for Class TimeLapseDlg .................................43
Figure 4.4.1 Scenario for Query Database by Plant_ID ..................46
Figure 4.4.2 Scenario for Display Time Lapse Distribution Map ....47
Figure 4.4.3 Printing Query Result ...........................................47
Figure 5.1  Basic Frame Window .........................................................54
Figure 5.2.1  Query Dialog – enter data ..............................................55
Figure 5.2.2  Query Dialog – select plant from list ..............................56
Figure 5.3  Query Message Box ..........................................................57
Figure 5.4  Display Table .....................................................................58
Figure 5.5  Display Plot ........................................................................59
Figure 5.6  Display Distribution Map .....................................................60
Figure 5.7.1  Times Lapse Period Dialog Box .......................................60
Figure 5.7.2  Display Times Lapse Distribution Map .............................61
Figure 6.1.1  Message Box .................................................................63
Figure 6.1.2  Table ..............................................................................64
Figure 6.1.3  Plot ................................................................................65
Figure 6.1.4  Distribution Map ...............................................................66
Figure 6.1.5  Time Lapse Distribution Map ............................................67
Figure 6.2.1  Message Box .................................................................68
Figure 6.2.2  Table ..............................................................................68
Figure 6.2.3  Plot ................................................................................69
Figure 6.2.4  Distribution Map ...............................................................70
Figure 6.2.5  Time Lapse Distribution Map ............................................71
Figure 6.3.1  Message Box .................................................................72
Figure 6.3.2  Table ..............................................................................73
Figure 6.3.3  Plot ................................................................................74
Figure 6.3.4 Distribution Map ........................................ 75
Figure 6.3.5 Query County Name ........................................ 75
Figure 6.3.6 Print Preview ............................................... 76
Figure 6.4 Invalid plant_id ............................................... 76
Chapter 1

Introduction

1.1 Purpose of the INVADERS Software Project

The purpose of the INVADERS software system is to build a computerized catalogue of plant names and an enhanced electronic atlas of weed distribution data. The software is designed for ease of use and might be considered to be a computerized companion to "Weeds of the West".

The INVADERS system provides weed regulatory and natural resource management agencies with an updatable database and data management software to support proactive weed management strategies and development of weed identification training programs. It also determines which alien weeds are most rapidly spreading over a multi-state region before they cause severe economic losses and environmental damage requiring perpetual control over large geographic areas.

Our present system for building awareness of potentially serious new weeds among vegetation management professionals, teaching identification of new weeds, regulatory listing, and developing control strategies for alien weeds is fundamentally reactive. We typically do not know which new exotics should be considered for active monitoring and aggressive management until the extent and
severity of the weed requires perpetual control of that weed over large areas of a state or multi-state region.

By not developing pro-active weed management strategies the costs of vegetation management increases by many orders of magnitude. In the past, land managers had not reacted until the degraded acreage was very large.

The INVADERS software is not another Geographic Information System. GIS is the appropriate tactical tool if you need to map the boundaries of a specific weed infestation and relate that infestation to topographic or environmental features. However, a GIS requires high end computer hardware, has a steep learning curve, very high data costs, and usually requires a specialized support staff. Strategic planning does not require GIS levels of detail. In fact, excessive details may impede strategic planning.

The INVADERS system is a strategic weed management tool. It is designed to support programmatic decision making. It allow vegetation program managers to view distribution data showing weed regional scale spread patterns over long time periods. INVADERS is very easy to learn, uses historic data with low spatial resolution (county presence or absence), and has a very low cost for the end user.[6]

1.2 Design Goals

The primary design goal of the INVADERS software/database system is to create
a practical and easy to use tool for managers of vegetation/weed programs. It is intended for natural resource program managers that do not have the time to learn to use complicated software packages, it was not designed for computer experts. The software features which help to meet that goal include:

** Ease of use, learning time is less than 20 minutes.

** Will run on entry level personal computers.

** A shareware program, distributed at cost, no commercial license fees.

** The database can be queried by using any published name for the weed, including Weed Science Society of America common names.

** The database includes all vascular plants in NW states, tells whether plant is exotic or native.

** Shows regulatory status of the plant: noxious weed or whether the plant is on a protected/sensitive species list.[6]

1.3 Potential User Group

Every vegetation program manager from county weed supervisor to the Federal land manager could have a copy of the INVADERS mapping software on their PC. The potential users include the following:

County Weed Control Departments (199 counties in the five state area)

State Departments of Agriculture - Weed Coordinator

State Extension Services - Noxious Weed Specialist

State Land Departments
State Fish, Wildlife, and Parks Departments

US Forest Service

   Region 1 and Region 6 Headquarters -

      Range Implementation Director

      Pesticide Coordinators

   Forest Supervisor Offices - Forest Planners and EIS Team Leaders

   Ranger Districts - Weed Program Leader

Bureau of Land Management

   BLM State Offices - Weed Scientist

   BLM Area Offices - Conservationist

USDA Agricultural Plant Health Inspection Service - State Officer in Charge

Federal Fish & Wildlife Service

University Faculty with -

   Courses on invasive plants, range management, & biological conservation

Researchers & Agencies Distributing Biological Control Agents for Weeds

Herbicide Companies


1.4 Some Possible Extensions

- Generate an "Alert List" of recently introduced weeds that are spreading rapidly across the region. An early warning list is an essential prerequisite for pro-active management.
• The Extension Service could determine which new weeds need to be included in ID training programs and bulletins, and where efforts for established weeds need to be increased.

• Each county weed supervisor or other agency weed manager could be provided with a county, approximate area, district, or forest specific list of exotic plants and sensitive native species.

• The Program can detect where a weed came from by plotting regional scale distribution patterns for sequential time intervals. Crop inspection, vehicle cleaning, and other regulatory actions could be used to reduce the transport of noxious weed seeds to new areas.

• Graphic plots of the cumulative number of counties reporting a weed over the year of the report combined with maps of the reported distribution suggest what type of regional scale strategic management might be most appropriate for that weed.

• Spread pattern plots, maps, and summary statistics and graphics derived from the database could be use to illustrate the extent and severity of the invasion process for decision makers who make budgetary and policy decisions.

• A shared database on weed distributions and spread rates could reduce the political aspects of listing weeds as noxious.

• Distribution maps help identify possible multi-agency cooperators for weed management projects.
• Taxonomic name questions are greatly simplified because the INVADERS software links old names (synonyms) to the currently accepted scientific name as well as the Weed Science Society common name.

• Each specific weed name could be linked to an "IPM Toolbox" which would provide a list of registered herbicides, biological agents, and references for control strategies using grazing, cultivation, and mechanical techniques.[6]

1.5 Status of INVADERS Database

Approximately 80,000 distribution records have been collected and put into the database. The most recent compiled and externally distributed version of the database is INVADERS Database Release 6.3 (Jan 1996). Some summary statistics of information in and/or calculated from Release 6.3 follow:

• 7,538 plant names for 6,113 species found in the 5 state project area (WA, OR, ID, MT, WY).

• 1,254 exotic plant names for 988 exotic species.

• 900 sensitive, threatened, or endangered plant species

• 16% of the plant species established outside of horticultural settings are exotic to North America.

• 76,115 distribution records covering 841 exotic species

• The distribution data spans 1875-1995 and was collected from the
following sources:

Dept of Agriculture, MT
Dept of Agriculture, OR
Dept of Agriculture, WY
USFS R01 ECODATA
Extension Service, ID
Extension Service, MT
Extension Service, OR
Extension Service, WA
F. Forcella study
U. of Idaho, Collection of Forestry (Moscow)
U. of Idaho herbarium (Moscow)
Madrono journal
U. of Montana herbarium (Missoula)
Montana State U. (Bozeman)
Montana Natural Heritage Program (Helena)
U. of Oregon herbarium (Eugene)
Oregon State U. herbarium (Corvallis)
Region 6 ecology plot data
Rocky Mountain Herbarium, U. of Wyoming (Laramie)
Rich Old extension records
Roger Sheley (MSU) 10/17/94 Weed Book draft
R. Sheley 1994 targeted weed survey data

Mike (Sherm) Karl Columbia River Basin 1995 weed survey data

U. of British Columbia herbarium (Vancouver)

Whitman College herbarium (Walla Walla, WA)

Willamette U. herbarium (Salem, OR)

Dept. of Forestry & Range Mgmt., WSU (Pullman)

Washington State U. herbarium (Pullman)

U. of Washington herbarium (Seattle)

Western Washington U. herbarium (Bellingham)

• 206 species declared noxious by state or federal governments (112 on state lists, the additional 94 are on only the federal list and not established in the region).

• 1.8% of the northwest flora are noxious.

• 11% of the introduced exotics are noxious.

• The number of exotics documented as established in each state are:[6]

  Washington  633
  Oregon 580
  Idaho  428
  Montana  485
  Wyoming  217
Chapter 2

Object Oriented Model

2.1 Object Oriented Design Definition

What is object-oriented design? Let’s look at Booch’s definition:

"Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design" [1]

The notation of object-oriented decomposition is very important. Classes and objects are used as the basic building blocks of the object-oriented design.

What is an object?

"An object has state, behavior, and identity, the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable."[1] The state of an object is determined by its static and dynamic properties. Examples of these properties are types of variables (static) and values of variables (dynamic). The behavior of an object represents its visible and testable activity. The identity of an object is a property that distinguishes the object from any other object.

What is a class?
“class is a set of objects that share a common structure and a common behavior.”[1]

The concepts of object and class are closely related, because each object is an instance of some class. However, there are also important differences between an object and a class. An object is a concrete entity that exists in time and space, but a class is an abstract description of the characteristics common to its objects. Another difference is that classes are mainly static, that is their existence, relationship, and semantics are fixed before execution of a program, but objects are created and destroyed dynamically during the program execution. From the designer’s point of view a class is an important entity, but class descriptions alone do not constitute the design of the system. Since the classes do not exist in isolation, it is necessary to identify the relationships among the classes and the objects in the system. This is a very important step in object oriented design, because these interactions define the overall behavior of the system.[5]

Booch gives six kinds of relationships that can be used to describe class/class, class/object, object/object interactions: association, inheritance, aggregation, using, instantiation and metaclass. Booch[1] gave us examples of these relationships and Votava[5] described them in an intuitive way:

Association refers to a bidirectional semantic dependency between two classes. For example, there is an association between students and courses offered by school, which in terms of cardinality is a many-to-many association, meaning
that each student is taking many courses and each course is attended by many students. Inheritance is a relationship among classes, in which a subclass is identified as one which shares all the structure and behavior of its superclass (or parent class), though the subclass may have additional properties not possessed by the parent. Aggregation defines the whole/part relationship between classes, where instance of one class (C₁) can be an attribute of another class (C₂). In this case class C₁ is the part of the class C₂ which represents the whole. A client-server interaction is depicted in the using relationship, in which one class is requesting services of another class. Each of the above is a class/class relationship, which means that if there is a relationship R between classes C₁ and C₂, then any object O₁ of C₁ and O₂ of C₂ have the same relationship R.

To add a higher level of abstraction and genericity, instantiation and Metaclass relationships are used. Instantiation is a relationship between a parametrized class (also called a generic class) and its instances. A parametrized class is an abstract class that must be instantiated before objects can be created, and so it serves as a template for other classes. Last of the relationships mentioned above is metaclass, which is a class whose instances are themselves classes. This relationship treats classes as objects that can be manipulated.[5]

There are two main tasks to be achieved by the object-oriented designer. First, the classes and objects must have been identified from the problem domain. Second, the relationships among the classes and objects must be identified. In the
terms of implementation, the classes and objects are called the key abstractions and the relationship structures are the mechanism of the design and implementation.[5]

2.2 Advantage of Object-Oriented Model

Why do we want to use object-oriented methodology to build this system? The alternatives to object-oriented methodology are traditional top-down structured design or data-driven design. Structured design methodology evolved to guide developers who were trying to build complex systems using algorithms as their fundamental building blocks while object-oriented methodology evolved to use classes and objects as basic building blocks.

The main benefits of the object-oriented model were listed by Booch as follows: First, the use of the object model helps us to exploit the expressive power of object-based and object-oriented programming languages. Next, the use of the object model encourages the reuse not only of software but of entire designs, leading to the creation of reusable application frameworks. Third, the use of the object model produces systems that are built upon stable intermediate forms, which are more resilient to change. Fourth, the object model reduces the risks inherent in developing complex systems. Finally, the object model appeals to the working of human cognition. The object-oriented systems are very natural.[1]
The object-oriented method views the system as a collection of cooperative objects that either cause the action or are the subjects upon which these operations act. The object-oriented method has the potential to yield smaller systems through the reuse of common mechanisms, thus providing an important economy of expression. The object-oriented method directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space.[1]

In this project, the major benefit of the object-oriented method comes from code reuse, namely: using an application framework. A framework is a collection of classes that provide a set of services that clients can use or adapt. A foundation class library is usually a domain-neutral framework, meaning that it applies to a wide variety of applications. We took the advantage of MFC (Microsoft Foundation Class Library) in this project. All of the other benefits as mentioned above are additional reasons for using object-oriented development over traditional structured development.

2.3 Notation

Before starting our discussion on the design phase of the INVADERS system, let’s introduce the notation that will be used in the following chapter. The main purpose of notation is to present the design in a manner that is more formal than a written description, easier understood than a source code listing, and more generic than a programming language. We will use four of the diagrams described by
Booch, namely:

- **Class Diagrams**
- **Object Diagrams**
- **State Transition Diagrams**
- **Interaction Diagrams**

### 2.3.1 Class Diagram Notation

Class diagram is used to show the existence of classes and their relationships in the logical view of a system[1]. Figure 2.1 shows the icon we will use to represent a class in a class diagram in the following chapters.

![Class Icon](image)

**Figure 2.1 Class Icon**

Since classes are related to each other in a system, we need some means to show the relationships among classes that were discussed above. These relationships are shown in the following figures.

Figure 2.2 “Class Relation – Association” shows class A associated with class B. The association icon connects two classes and denotes a semantic connection.
Figure 2.2 Class Relation -- Association

Figure 2.3 "Class Relation – Use" shows that class A uses class B. In another words, class A uses the services supplied by class B. The “using” icon denotes a client/supplier relationship, and appears as an association with an open circle at the end denoting the client.[1] This relationship indicates that the client in some manner depends upon the supplier to provide certain services.

Figure 2.3 Class Relation -- Use

Figure 2.4 “Class Relation – Has” shows class B is a component of class A. The “has” icon denotes a whole/part relationship ( the “has a” relationship, also known as aggregation. ), and appears as an association with a filled circle at the end denoting the aggregate. The class at the other end denotes the part whose instances are contained in the aggregate object.[1]
2.3.2 Object Diagram Notation

Object diagrams are used to show the existence of objects and their relationships in the logical design of the system. In other words, object diagrams are used to represent a set of possible interactions among objects in the system. The icon for object is shown in Figure 2.5.

Because objects communicate via sending messages, the relationship shows the direction of the message passing using an arrow, labeled with the message been sent. Optionally, sequence numbers can be shown on the arc in diagrams that try to indicate the relative sequence of messages in a particular scenario. Messages with the same sequence numbers are unordered relative to each other; messages with lower sequence numbers are dispatched before messages with higher sequence numbers.
2.3.3 State Transition Diagram Notation

The state of an object represents the cumulative results of its behavior. An event is some occurrence that may cause the state of a system to change. *State transition diagrams* are used to describe dynamic behavior of individual objects. They show the external events that cause an object to change its internal state, and the internal actions that result from this change. Each state should have a name that is unique for all the states within this particular object, and an optional list of actions associated with this state. A *state transition* (Figure 2.6) represents a change of state in an object. The change is usually triggered by some event, and subsequently an action is performed which changes the internal state of the object.

Figure 2.6 STD icons show the state transition diagram icons. The left icon is the beginning icon. The middle box icon denotes a state icon which has name and actions. The right most icon is the ending icon. The arrows used in the STD is showing the event/action as below.

![STD Icons](image)

**Figure 2.6 State Transition Diagram Icon**
2.3.4 Interaction Diagram Notation

An *Interaction diagram* is used to trace the execution of a scenario in the same context as an object diagram. Indeed, an interaction diagram is simply another way of representing an object diagram. The advantage of using an interaction diagram is that it is easier to read the passing of messages in relative order. The advantage of using an object diagram is that it scales well to many objects with complex invocations, and permits the inclusion of other information, such as links, attribute values, roles, data flow, and visibility.[1]

The main difference between an object diagram and an interaction diagram is that in an interaction diagram it is easier to see the exact order of messages exchanged between objects. The reason for using both diagrams is that interaction diagrams do not scale well when the number of objects increases – object diagrams are much more readable in this case, because they omit the ordering details.[5] Figure 4.4.1 in this document shows an example of an interaction diagram.
Chapter 3

System Analysis

3.1 System Requirements

The INVADERS system shall provide a user-friendly interface for accessing the INVADERS database and getting the query results. Specifically, it must:

- allow users to enter either the plant ID, the common name or the Latin binomial to query against the database
- allow users to select the geographic region (five northwest states and their counties accordingly)
- allow users to specify a query stop year
- allow users to select a plant from the exotic plant list to query
- allow users to select a county to query the exotic plant in all of its contiguous counties

The system shall also provide the query result in the following forms:

- table of records of the plant in the database with the title showing how many records there are in the database, the family name, common name and scientific name of this plant and whether the plant is native to the Northwest
- plot of the cumulative number of counties infested by the plant versus the
time period.

- distribution map of Northwest region showing counties infested by the plant

The system shall also provide a zoom-in distribution map when the user selects a specific state to query against the database. The system should display the county name when the users query that name against the distribution map. The system also needs to display the plant spread trend in a time lapse period distribution map format. The system shall also allow users to print out all the query results. For example, if the user sets the Plant ID to “ABRMELVF”, the query stop year to “1997” and the region to “All Northwest”, and queries the database, the result of this query would be:

![Query Result Message Box](image)

**Figure 3.1 Query Result Message Box**

All of the records for this plant can be displayed in tabular form as shown in the Following:

*Query spans from 1972 to 1997 for the NW region.*
Database contains 2 distribution records of this species. (for your current query: region and stop year)

The family name is: Nyctaginaceae

The common name is: white abronia

The scientific name is: Abronia mellifera (ABRMELVF)

<table>
<thead>
<tr>
<th>#</th>
<th>year</th>
<th>county</th>
<th>state</th>
<th>sp-id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1972</td>
<td>Grant</td>
<td>WA</td>
<td>ROLD-00000004062</td>
</tr>
<tr>
<td>2</td>
<td>1986</td>
<td>Gem</td>
<td>ID</td>
<td>ROLD-0000001542</td>
</tr>
</tbody>
</table>

The user can also view a plot of the cumulative number of counties infested versus the time period:

Figure 3.2 Query Result -- Plot

The distribution map of this plant for the Northwest region would look like figure 3.3. All counties infested by this plant will be marked.
3.2 System Analysis

3.2.1 Defining the Boundaries of the Problem

System analysis suggests that there are several major functional activities in the system:

- **Query Condition Input**  Responsible for accepting query condition input from the user
- **Query Against Database**  Responsible for querying the database according to the user’s query condition
- **Display Table**  Responsible for displaying query result in tabular form
- **Display Plot**  Responsible for calculating the number of
We can see that our system retrieves large volumes of data. So most of our design should involve decisions about declarative knowledge (what entities exist, what they mean, what is the current status) rather than procedural knowledge (how things happen). The key of our design will be found in the central concerns of object-oriented development: the key abstractions that form the vocabulary of the problem domain and the mechanisms that manipulate them. Our system is therefore designed in order to incorporate these functions.

Before we start identifying classes in our system, let’s look at the architecture of our framework, i.e., Microsoft Foundation Class Library (MFC). The classes in MFC make up an “application framework” — the framework on
which you can build an application for windows. The framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton at a very general level. The most important classes for building Windows application are the application class, the frame window class, the document class, and the view class. Since we deal with database tables in our system, the database classes are also very important.

- **Application Class**

The application class encapsulates the initialization, running and termination of an application for Windows. An application built on the framework must have one (and only one) object of a class derived from class `CWinApp`.

- **Frame Window Class**

The Frame Window class `CFrameWnd` provides the functionality of a Windows frame window which contains and manages the view. When an application runs under Microsoft Windows, the user interacts with documents displayed in frame windows. A frame window has two major components: the frame and the contents that it frames. An application built on the framework might contain an application-derived frame window class.

- **View Class**

The `CView` class provides the basic functionality for application-derived view
classes. A view is attached to the document and acts as an intermediary between the document and the user: a view renders an image of the document on the screen or printer and interprets user input as operations upon the document. A view is a child of a frame window.

The framework uses a frame window to contain a view. The drawing and other user interaction with the document occurs in the view’s client area. The frame window provides a visible frame around a view, complete with a caption bar and standard window controls such as a control menu, buttons to minimize and maximize the window, and controls for resizing the window. The “contents” consist of the window’s client area, which is fully occupied by a child window—the view. Figure 3.4 shows the relationship between a frame window and a view.

![Diagram of frame window and view relationship]

**Figure 3.4 Relationship Between Frame window and View**

- Document Class
The parts of the framework most visible both to the user and to the designer are the document and view. Most of the work in developing an application with the framework goes into writing application's own document and view classes which are derived from `CDocument` and `CView` accordingly.

The `CDocument` class provides the basic functionality for application-defined document classes. A document is used to manage an application's data. A view is used to display the data and take users input. A frame window puts a frame around a view. Codes that define and manipulate data reside in the document class. Codes that display the data and interprets users input reside in the view class. Figure 3.5 shows the relationship between document, view and data.

![Figure 3.5 Relationship of Document-View-Data](image)

- **Database Class**

In the MFC database programming model, the `CRecordset` class provides basic functionality for application-defined recordset classes. A recordset object
represents a set of records selected from a data source. For every distinct table you wish to access, you must define a class derived from CRecordSet. You make queries and updates via recordset object.

We now begin to analyze the classes in the INVADERS system by using the architecture of application framework (MFC). First, we need a class to represent the INVADERS database application. An object of this class will denote an instance of the application. An object of this class will also provide member functions for initializing the application and for running the application. Second, we need a class to represent a set of records selected from the database table. We decide that this class will be responsible for keeping track of records obtained from the current query. Since there are a total of nine database tables in this system, we might want to derive nine child classes from this class to represent data sets accordingly. Database operations must be supported by this class. Typically, the operation open/close will be used to open or close the data set. Operation getstatus will be used to get current status of the data set. Operation move will be used to navigate forward or backward the data set. Operation update will be used to edit values of records in the data set and write the modification back to database (in case we need to modify the database). Operation find will be used to search for records according to some conditions. We also need to a sorter and a filter to sort and filter the record set. Third, we need to derive our own document, view and frame window class from framework, namely: class CDocument, class CView and class CFrameWnd. Fourth, we need a class called
query-dialog to accept query condition input. We need query dialog to take input of Latin Binomial, Common Name, Plant-ID, Query Region and Query Stop Year. The dialog would also contain an exotic plant list that users can choose from. Fifth, we need a class called time-lapse-dialog to represent our time lapse value input dialog. We need this dialog to take the input value of a time lapse period value. Last, we need a list class to serve as a collection of plant names so that users can choose a plant from that list.

3.2.2 Database Models

A database “is a repository for stored data. In general, it is both integrated and shared. By ‘integrated’ we mean that the database may be thought of as a unification of several otherwise distinct data files, with any redundancy among those files partially or wholly eliminated... By ‘shared’ we mean that individual pieces of data in the database may be shared among several different users”[3]. With centralized control over a database, “inconsistency can be reduced, standards can be enforced, security restrictions can be applied, and database integrity can be maintained.”[3]. In the INVADERS system, the database model is relational. As Booch mentioned: “Relational database technology is much more mature, available across a wider variety of platforms, and often more complete, offering solutions to the issues of security, versioning, and referential integrity. The relational database model has indeed proven to be very popular.”[1]

The basic elements of a relational database “are tables in which columns
represent things and the attributes that describe them and rows represent specific instances of the things described... The model also provides for operators for generating new tables from old, which is the way users manipulate the database and retrieve information from it"[4].

The INVADERS database system is combined with several tables. The Meta table, is a dictionary of plant names found in the distribution data (data about data is called meta data). The Meta table specifies accepted names, old synonyms, sub-divisions, and common names for plants in the Northwest. It also includes information on the origin and federal status of the plant. Meta table uses two reference tables: Plant table and Fam_dict table (family dictionary).

The plant table is a reference table for quickly answering questions like: is the plant in the database? or how many distribution records are there for a plant? The Fam_dict table serves as a family name dictionary.

Dstrb table keeps distribution data of plants in the Northwest region. Dstrb table also uses two reference tables: Sp_id and County. The Sp_id table stores a list of every specimen_id in the Dstrb table. The County table maps a county of a state to a unique county code.

Noxlist table stores all the noxious plant of the Northwest. Sensb table stores all the sensitive plants of the Northwest. The complete formal description of these database tables is covered in Appendix B.
Chapter 4

System Design

We have already identified classes in our analysis phase. However, we have not described more detailed descriptions of the internal structure of our system. This chapter will describe the classes and objects that were identified during our analysis and their relationships. The formal description of the key classes is covered in Appendix A.

4.1 Class Diagrams

Figure 4.1 provides a class diagram that express the architecture for the INVADERS system. Most of the classes here have already been discovered during analysis.

![INVADERS System Class Diagram](image)
Let's look at the major class relationships in Figure 4.1. This diagram shows that in our system, the document class contains all database classes. The application class uses the view class, the document class and the frame window class. The frame window class has class view. The view class uses the document class, the query dialog class and the time lapse dialog class. The query dialog class uses class List.

4.2 Object Diagram

An object diagram is used to show the existence of objects and their relationships in the system. Let's look at a general scenario: the user invokes the INVADERS system and presses the query dialog button. Then the user inputs the query condition and presses OK button. After the system returns a message box of the query result, the user can choose to display the query result in the form of a table, distribution map, plot or/and time lapse distribution map.

Figure 4.2 Shows the partial object interactions of this scenario. The first object: InvaderApp sends 1: Create() message to object Frame to create a main frame window which contains menu bar, toolbar and view area. Then 2: OnQueryButton() sends message to object QueryDlg to display a query dialog. Based on the order, then class QueryDlg object either sends 3': OnCancel() to itself to destroy the dialog or send 3: OnOk() to verify the query conditions just selected by the user. If query conditions are valid, then object QueryDlg sends 4: OnQuery() to object View to query the database object. Object View then sends
the messages 5, 6, 7, 8, 9, 10: Query() to database object CountySet, FamdictSet, SensbSet, MetaSet, NoxlistSet and DstrbSet. Object DstrbSet will send the message 11: DispResult() to object View to display the query result as soon as the query is done. Object Frame will send message 12: DispTimeDlg() to object TimeLapseDlg to display time lapse dialog when the user presses the DispTimeLapseMap button or send the message 12': UpdateView() to object View to display corresponding results when the user presses DispTable, DispDstrbMap or DispPlot button. Object TimeLapseDlg will send message 13: UpdateView() to object View to update the view.
4.3 State Transition Diagrams

The state of an object represents the cumulative results of its behavior. An event is some occurrence that may cause the state of a system to change. A state transition diagram is used to show the state space of a given class, the events that cause a transition from one state to another, and the actions that result from a state change. A single state transition diagram represents a view of the dynamic model.
of a single class or of the entire system[1]. Let’s look at state transition diagram for some important classes. Figure 4.3.1 shows the state transition diagram for class DstrbSet. Figure 4.3.2 shows the state transition diagram for class QueryDialog. Figure 4.3.3 shows the state transition diagram for class View. Figure 4.3.4 shows the state transition diagram for class TimeLapseDlg.

![State Transition Diagram for Class DstrbSet](image1)

Figure 4.3.1 STD for Class DstrbSet

![State Transition Diagram for Class QueryDialog](image2)

Figure 4.3.2 STD for Class QueryDialog
4.4 Interaction Diagrams of Key Scenarios

We begin by enumerating a number of primary scenarios:

- Querying database by plant id with specified query stop year and geographic
region

- Querying database by common name with specified query stop year and geographic region
- Querying database by Latin binomial with specified query stop year and geographic region
- Querying database by selecting a plant from exotic plant list with specified query stop year and geographic region
- Specifying a county name and querying exotic plant conditions of its neighboring counties
- Displaying query result in the form of record table
- Displaying query result in the form of distribution map
- Displaying query result in the form of plot
- Displaying query result in the form of time lapse distribution map
- Printing / printing preview the query result

For each of these primary scenarios, we can envision a number of secondary ones:

- Entering wrong plant_id
- Entering wrong common name
- Entering wrong Latin binomial
- Entering query condition in more than one place and they are not matched (for example, enter plant-id and common name concurrently but they are not a match)
- Selecting a wrong geographic region
• Specifying a wrong query stop year
• Selecting a wrong plant name from exotic plant list
• Canceling query

We now examine a number of these scenarios in order to illuminate the behavior of the system. Query database by plant id with specified query stop year and geographic region is one of the primary functions points of the INVADERS system.

4.4.1 Query Database by Plant_ID

Figure 4.4.1 illustrates this scenario. Here we see that the user enters plant_id and query stop year and selects a query region. The plant_id should be an eight character string. Since the user entered an invalid plant_id, INVADERS system prompt the user to change it. The user re-submitted the plant_id and click “OK” button of query dialog, the query dialog will then submit this request to view. The view will parse the query conditions and send the query request to document. The document in turn sends the query request to record set and record set queries the actual database tables. The database tables return query result to record sets and record sets send a message to view to display the result.

4.4.2 Display Time Lapse Distribution Map

Figure 4.4.2 demonstrates a display result in the form of a time lapse distribution
map, another critical system behavior. For resource reasons, the time lapse period must not be too small. The minimum acceptable value is five. Since the user entered a non-valid time lapse period, the system prompted user to reenter it. Here we can see that the database is not involved in this scenario. The user requests time lapse distribution map by clicking button on the frame window. Frame window then displays the time period dialog to prompt user to enter the time period. If that value is valid, the time period dialog sends a message to view to display the result.

![Diagram of query database by plant ID](image)

**Figure 4.4.1 Scenario for Query Database by Plant_ID**
Object-oriented Design and Implementation of INVADERS System

4.4.2 Scenario for Display Time Lapse Distribution Map

4.4.3 Printing

Now let’s examine another key scenario, printing the query result. Figure 4.4.3 illustrates this scenario.

Figure 4.4.3 Printing Query Result
The user submits a printing request by clicking the print button or menu on the frame window. The frame window then displays the print dialog to select the printing properties. The user requests print preview, then print dialog sends print preview message to view and displays print result on the screen. After the user confirms the printing request, the view will send the print command to the printer which prints out the result.
Chapter 5

Implementation

I have implemented the system by using Microsoft Visual C++. MFC (Microsoft Foundation Class Library) has been used to implement the user interface. ODBC (Open Database Connectivity) has been used to link the database tables to corresponding database class objects. But why C++?

When deciding the implementation of the project, we had to first choose the implementation language. An alternative to C++ is Java. Using Java has many advantages. For example, we could use Java to create a web-browser applet so that the users can run our program through their web browser. This makes database maintenance easier because we don’t have to distribute the database along with our application. Also, since Java is system independent, the applet would run on Mac, Unix and Windows platforms. However, when we started this project, many INVADERS users were still using entry-level computers and also many of them did not have network connections. Further, we didn’t have a machine which was sufficient to serve as a web server at that time. So according to the INVADERS primary user Peter M. Rice of DBS, University of Montana, we decided to implement the INVADERS program by using C++. Indeed, the INVADERS web-browser version is Peter’s next project.

5.1 Understanding ODBC
ODBC (Open Database Connectivity) is a call-level interface that allows applications to access data in any database for which there is an ODBC driver. ODBC provides an API that allows applications to be independent of the source database management system (DBMS). ODBC is the database portion of the Microsoft Windows Open Services Architecture (WOSA), an interface which allows Windows-based desktop applications to connect to multiple computing environments without re-writing the application for each platform. The following are components of ODBC:

- **ODBC API** is a library of function calls, a set of error codes, and a standard Structured Query Language (SQL) syntax for accessing data on DBMSs.

- **ODBC Driver Manager** is a dynamic link library (ODBC.DLL) that loads ODBC database drivers on behalf of an application. This DLL is transparent to applications.

- **ODBC Database drivers** are one or more DLLs which process ODBC function calls for specific DBMSs. Since we are using Paradox for Windows as our database system, we must use ODBC Paradox driver.

- **ODBC Cursor Library** is a dynamic link library (ODBCCURS.DLL) that resides between the ODBC Driver Manager and the driver that handles scrolling through the data.

- **ODBC Administrator** is a tool used for configuring a DBMS to make it
available as a data source for an application. Applications achieve independence from DBMSs by working through an ODBC driver written specifically for a DBMS rather than working directly with the DBMS. The driver translates the calls into commands its DBMS can use and makes it available for a wide range of data sources. The ODBC drivers manage the connections to the data source, and SQL is used to select records from the database. A data source is a specific instance of data hosted by some database management system (DBMS). The information required to access the data and the location of the data source can be described using a data source name. To work with database classes derived from class CDatabase, we must use the Open Database Connectivity (ODBC) Administrator to configure the data source. In the INVADERS database system, there are nine tables. So we should have nine data sources active in our application at one time, each represented by an object of database class derived from class CDatabase.

5.2 The Role of SQL in ODBC

Structured Query Language (SQL) is a way to communicate with a relational database that lets you define, query, modify, and control the data. Using SQL syntax, you can construct a statement that extracts records according to criteria you specify. SQL statements begin with a keyword "verb" such as CREATE or SELECT. It is a very powerful language; a single statement can affect an entire
The database classes are implemented with ODBC, which uses SQL in a call-level interface rather than embedding SQL commands in the code. ODBC uses SQL to communicate with a data source through ODBC drivers. These drivers interpret the SQL and translate it, for use with a particular database format.

The object of derived database classes is to use ODBC to communicate with a data source, and ODBC retrieves records from the data source by sending SQL statements. A database class object constructs a SQL statement by building up the pieces of an SQL statement into a \texttt{CString}. The string is constructed as a SELECT statement, which returns a set of records. Then the database object calls ODBC to send an SQL statement to the data source, the ODBC Driver Manager loads the appropriate ODBC driver. The Driver Manager passes the statement to the ODBC driver, which in turn sends it to the underlying DBMS. The DBMS returns a result set of records, and the ODBC driver returns the records to the application.

5.3 Understanding MFC

MFC stands for Microsoft Foundation Class Library. There are several subsets of classes within Microsoft Foundation Class Library. The subsets used by the INVADERS system are: Application Architecture Classes, Visual Object Classes,
General Purpose Classes and Database Classes. Following are a list of some important foundation classes used in this project.

*Application Architecture Classes*

*CWinApp* --- base class from which a windows application is derived

*CDocument* --- base class for application derived document class

*CView* --- base class for application derived view class

*Visual Object Classes*

*CFrameWnd* --- base class for application derived frame window

*CDialog* --- base class for application derived dialog

*General Purpose Classes*

*CFile* --- base class for file classes

*CArchive* --- class of binary stream object which serves as an intermediate between permanent storage and buffered data.

*Database Classes*

*CRecordSet* --- base class for application derived record set.

5.4 **User Interface**

5.4.1 **Basic Frame Window**

Figure 5.1 shows a basic frame window of the INVADERS database application. The frame window consists of window frame, menu bar, tool bar, scroll bar, status bar and a central display area. There are five menus on the menu bar: File,
Query, Result, View and Help. Under File menu, there are four submenus: Print, Print Preview, Print Setup and Exit. The "Query" menu is used to trigger the query dialog. The "Result" menu is used to display the query result in the form of table, plot and distribution map. The "View" menu is used to display previous page or next page in case there are multiple pages of result and toggle on/off the display of toolbar and status bar. From "Help" menu, you can get basic project information. The Buttons on the toolbar are shortcuts to menu items.

![Figure 5.1 Basic Frame Window](image)

**Figure 5.1 Basic Frame Window**

### 5.4.2 Query Dialog

Figure 5.2.1 and 5.2.2 show the query dialog which is used to input user query conditions. The user can either choose to enter query data or to select a plant from
an exotic plant list to query the database. The user can also select a geographic region; then the county drop down combo box will change its contents accordingly. The user can also change the value of query stop year (by default, it equals current year). If the user selects “Select List” button, the Latin binomial, common name and plant_id fields will become disabled and a list of exotic plants will be displayed in the exotic plant list box. Please see Figure 5.2.2 for this use.

![INVADERS DATABASE QUERY](image)

Figure 5.2.1 Query Dialog – enter data
Figure 5.2.2 Query Dialog – select plant from list
5.5 Query Result Display

5.5.1 Message Box

![Query Message Box]

Figure 5.3 Query Message Box

Figure 5.3 shows the message box which displays the result of a query. From the message box we can find the family name, common name and scientific name of this plant, how many records in the database correspond to this plant, and whether its native or exotic to the Northwest.
5.5.2 Display Table

Query spans from 1972 to 1997 for the NW region.
Database contains 2 distribution records of this species.
(For your current query: region and stop year).
The family name is: Nyctaginaceae
The common name is: white abronia
The scientific name is: Abronia mellifera (ABRMELUF)
This plant is Native to North America.

<table>
<thead>
<tr>
<th>#</th>
<th>year</th>
<th>county</th>
<th>state</th>
<th>sp_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1972</td>
<td>Grant</td>
<td>WASHINGTON</td>
<td>R0LD_0000004062</td>
</tr>
<tr>
<td>2</td>
<td>1986</td>
<td>Gem</td>
<td>IDAHO</td>
<td>R0LD_0000001542</td>
</tr>
</tbody>
</table>

Figure 5.4 Display Table

Figure 5.4 shows records within the database for this plant.
5.5.3 Display Plot

![cumulative number of counties infested](image)

**Abronia mellifera white abronia SPREAD:**
found 2 counties with records of ABRMELVF

Figure 5.5 shows number of counties infested by this plant versus time period.

5.5.4 Display Distribution Map

Figure 5.6 shows the distribution map of the plant for the Northwest region. From the distribution map, we can see that the counties infested by the current plant are marked with color green. Also, if you move the mouse pointer to a county on that map and double click the left button, the county name and state will be displayed on the screen.
Figure 5.6  Display Distribution Map

5.5.5  Display Time Lapse Distribution Map

Figure 5.7.1 Times Lapse Period Dialog Box

Figure 5.7.1 shows the dialog box which allows you to enter a time lapse period.
Figure 5.7.2 Display Times Lapse Distribution Map

Figure 5.7.2 shows the time lapse distribution map of this plant. In order to view the time lapse distribution map of the plant, you must specify a time period.
Chapter 6

Testing

6.1 Testing and Results

In object-oriented architecture, system testing involves testing the system as a whole. Testing should focus upon the system’s external behavior; a secondary purpose of testing is to push the limits of the system in order to understand how it fails under certain conditions.

I designed several test cases to test the INVADERS system as a whole. The INVADERS system is used to query the database systems by inputting query conditions. The users can input either Latin binomial, common name or plant_id. The users can also select geographical region and select a plant from the exotic list to query the database. Therefore, my test cases should cover all of these possibilities in order to test the whole system.

Case 1.

Query database by:

1. `plant_id = "ABRMELVF"`
2. `region = "All Northwest"`
3. `county = "All County"`
4. `query stop year = 1997`
Query Result:

Figure 6.1.1  Query Message Box

![Query Message Box](image)

Invaders (Windows Version)

Database contains 2 distribution records of this species.

(For your current query: region and stop year).

The family name is: Nyctaginaceae

The common name is: white abronia

The scientific name is: Abronia mellifera [ABRMELVF]

This plant is Native to North America.
Figure 6.1.2  Table:

Welcome to Invaders Windows Version 1

Query spans from 1972 to 1997 for the NW region.
Database contains 2 distribution records of this species.
( For your current query: region and stop year ).
The family name is: Nyctaginaceae
The common name is: white abronia
The scientific name is: Abronia mellifera ( ABRMELUF )
This plant is Native to North America.

<table>
<thead>
<tr>
<th>#</th>
<th>year</th>
<th>county</th>
<th>state</th>
<th>sp_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1972</td>
<td>Grant</td>
<td>WASHINGTON</td>
<td>ROLD_0000004062</td>
</tr>
<tr>
<td>2</td>
<td>1986</td>
<td>Gen</td>
<td>IDAHO</td>
<td>ROLD_0000001542</td>
</tr>
</tbody>
</table>
Figure 6.1.3  Plot

Abronia mellifera white abronia SPREAD:
found 2 counties with records of ABRMELVF
Figure 6.1.4  Distribution Map

Abronia mellifera (white abronia) for 1972–1997 for the NORTHWEST region
Figure 6.1.5  Time Lapse Distribution Map

Case 2.

Query database by:

1. common name = “velvetleaf”

2. region = “MONTANA”

3. county = “All County”

4. query stop year = “1996”
Query Result:

Figure 6.2.1 Message Box

![Invaders (Windows Version) Message Box](image1)

Database contains 24 distribution records of this species. (For your current query: region and stop year)
The common name is: velvetleaf
The scientific name is: Abutilon theophrasti (ABUTHEVF)
This plant is Exotic to North America.
It is also a noxious species. in: OR, WA

Figure 6.2.2 Table

![Invaders Windows Version Table](image2)

Query spans from 1956 to 1996 for the NW region.
Database contains 24 distribution records of this species. (For your current query: region and stop year).
The common name is: velvetleaf
The scientific name is: Abutilon theophrasti (ABUTHEVF)
This plant is Exotic to North America.
It is also a noxious species. in: OR, WA

<table>
<thead>
<tr>
<th>#</th>
<th>year</th>
<th>county</th>
<th>state</th>
<th>sp_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1956</td>
<td>Missoula</td>
<td>MONTANA</td>
<td>MONTU0000005206</td>
</tr>
<tr>
<td>2</td>
<td>1961</td>
<td>Big Horn</td>
<td>MONTANA</td>
<td>FFL0000000002</td>
</tr>
<tr>
<td>3</td>
<td>1961</td>
<td>Big Horn</td>
<td>MONTANA</td>
<td>MONT0000005250</td>
</tr>
<tr>
<td>4</td>
<td>1961</td>
<td>Judith Basin</td>
<td>MONTANA</td>
<td>FFL0000000005</td>
</tr>
<tr>
<td>5</td>
<td>1961</td>
<td>Judith Basin</td>
<td>MONTANA</td>
<td>MONT0000005853</td>
</tr>
<tr>
<td>6</td>
<td>1964</td>
<td>Dawson</td>
<td>MONTANA</td>
<td>FFL0000000003</td>
</tr>
<tr>
<td>7</td>
<td>1964</td>
<td>Dawson</td>
<td>MONTANA</td>
<td>MONT0000061152</td>
</tr>
<tr>
<td>8</td>
<td>1968</td>
<td>Roosevelt</td>
<td>MONTANA</td>
<td>FFL0000000006</td>
</tr>
<tr>
<td>9</td>
<td>1968</td>
<td>Roosevelt</td>
<td>MONTANA</td>
<td>MONT0000064320</td>
</tr>
<tr>
<td>10</td>
<td>1970</td>
<td>Fallon</td>
<td>MONTANA</td>
<td>FFL0000000004</td>
</tr>
<tr>
<td>11</td>
<td>1970</td>
<td>Fallon</td>
<td>MONTANA</td>
<td>MONT0000065741</td>
</tr>
</tbody>
</table>
Figure 6.2.3  Plot

Abutilon theophrasti velvetleaf SPREAD:
found 14 counties with records of ABUTHFVF
Figure 6.2.4 Distribution Map

Abutilon theophrasti (velvetleaf) for 1956–1996 for the MONTANA region
Figure 6.2.5  Time Lapse Distribution Map

Case 3.

Query database by:

1. region = "WASHINGTON"
2. county = "Benton"
3. select "Russian knapweed" from exotic list
4. query stop year = 1997
Result:

Figure 6.3.1  Message Box

Invaders (Windows Version)

Database contains 12 distribution records of this species.
(For your current query: region and stop year.)
The family name is: Asteraceae
The common name is: Russian knapweed
The scientific name is: Acropilum repens (CENREPVF)
This plant is exotic to North America.
It is also a noxious species in: MT, ID, OR, WA, WY.
Welcome to Invaders Windows Version!

Query spans from 1926 to 1987 for the NW region.
Database contains 12 distribution records of this species. (for your current query: region and stop year).
The family name is: Asteraceae
The common name is: Russian knapweed
The scientific name is: *Acroptilon repens* (CENREPUF)
This plant is exotic to North America.
It is also a noxious species in: MT, ID, OR, WA, WV

<table>
<thead>
<tr>
<th>#</th>
<th>year</th>
<th>county</th>
<th>state</th>
<th>sp_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1926</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000041081</td>
</tr>
<tr>
<td>2</td>
<td>1929</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000065611</td>
</tr>
<tr>
<td>3</td>
<td>1929</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000065664</td>
</tr>
<tr>
<td>4</td>
<td>1938</td>
<td>Benton</td>
<td>Washington</td>
<td>OSC___0000037580</td>
</tr>
<tr>
<td>5</td>
<td>1938</td>
<td>Benton</td>
<td>Washington</td>
<td>OSC___0000037580</td>
</tr>
<tr>
<td>6</td>
<td>1938</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000221282</td>
</tr>
<tr>
<td>7</td>
<td>1950</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000205318</td>
</tr>
<tr>
<td>8</td>
<td>1950</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000205602</td>
</tr>
<tr>
<td>9</td>
<td>1952</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000242205</td>
</tr>
<tr>
<td>10</td>
<td>1969</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000320502</td>
</tr>
<tr>
<td>11</td>
<td>1986</td>
<td>Benton</td>
<td>Washington</td>
<td>WS___0000300478</td>
</tr>
<tr>
<td>12</td>
<td>1987</td>
<td>Benton</td>
<td>Washington</td>
<td>ROld_0000000397</td>
</tr>
</tbody>
</table>
Figure 6.3.3  Plot

Acroptilon repens Russian knapweed SPREAD:
found 1 county with record of CENREPVF
Figure 6.3.4  Distribution Map

![Distribution Map Image]

Figure 6.3.5  Query County Name

![Query County Name Image]
Case 4.

Invalid query conditions

plant_id = "abrmel"

Figure 6.4  Invalid plant_id
6.2 Test Cases Evaluation

Case 1

Case 1 is designed to test the ability of the INVADERS system to query the database by default query stop year, all Northwest region and plant_id. The test result shows that our system succeeds in this case.

Case 2

Case 2 is designed to test the ability of the INVADERS system to query the database by common name and also filter the result by specifying a query region (MONTANA) and a query stop year (1996). The test result shows that our system succeeds in this case.

Case 3

Case 3 is designed to test the ability of the INVADERS system to query the database by selecting a plant from the exotic plant list and filter the result by specifying a county of a state. The test result shows that our system succeeds in this case.

Case 4

Case 4 is designed to test the ability of the INVADERS system to remind the user if there is an error in input plant_id. The test result shows that our system is able to do this.
User Interface

The user interface of the INVADERS system complies with a standard Windows application interface and is quite simple. For each form of result, there is a corresponding menu/button. Thus the learning time is very short. I had the chance to have several users test the program and the learning time was no more than twenty minutes.
Chapter 7

Discussion and Conclusion

7.1 General Discussion

In general, I found that the INVADERS database system was able to meet the design goals. The INVADERS database system can provide useful information for the weed or vegetation manager to make programmatic decisions. The interface of this system is user friendly. However, the system was not able to tell if the user entered an incorrect common name or plant_id or Latin binomial. The system can only signify to the user that such a plant does not exist in the database. If our system had this capability, we might save user’s query time. We can also see that our database has some redundant information among different tables. For example, table noxlist.db contains genus, species, common name information of a noxious plant. However, this information could be retrieved from meta.db as well.

7.2 Suggestion For Future Work

Following are some suggestions for evolving the INVADERS system:

1. Incorporating a dictionary database table for plant_id, common name and Latin binomial. By searching this table, the system could inform the user if an incorrect query condition was entered.
2. Restructuring some database tables so that there is no redundant information.

3. Another interesting improvement of the INVADERS system is to implement the system into a JAVA applet. By using an applet, users could access our database through their favorite web browser without actually installing the whole database on their local machine.

7.3 Summary

Currently the INVADERS system meets the design goals and can be used by a weed manager to obtain useful information. The object-oriented analysis and design method was used to build this system. Microsoft Visual C++ was used as an implementation tool, ODBC was used to connect the database tables to the application, and SQL was used to actually query the database. From the system we can see that object-oriented analysis and design was successful in this process.
Appendix A - Class Specifications

Class InvadersApp::public CWinApp
{
  public:
  InvadersApp();
  ~InvadersApp();
  void OnHelp();
  BOOL InitInstance();
  protected:
    CStringAppName;
    HINSTANCE hCurrentInstance;
};

Semantics:
The instance of class InvadersApp is used to represent our INVADERS application. It is inherited from class CWinApp.

- InvadersApp: class constructor
- ~InvadersApp: class destructor
- OnHelp: display help dialog
- InitInstance: instance initialization

Class CountySet: public CRecordSet
{
  public:
    CountySet();
    ~CountySet();
    CString GetCountyName() { return county_name; }
    CString GetCountyCode() { return county_code; }
  protected:
    CString county_code;
    CString county_name;
};

Semantics:
Class CountySet is used to represent a set of records selected from county table. It is inherited from class CRecordSet.

- CountySet: class constructor
- ~CountySet: class destructor
- GetCountyName: It is used to get the county name of current record.
• GetCountyCode: It is used to get the county code of current record.

**Class FamdictSet: public CRecordSet**

```
public:
    FamdictSet();
    ~FamdictSet();
    CString GetGenus() { return Genus; }
    CString GetFamily_name() { return Family_name; }
    CString GetData_enterer() { return Data_enterer; }
    CTime GetEntry_date() { return Entry_date; }

protected:
    CString Genus;
    CString Family_name;
    CString Data_enterer;
    CTime Entry_date;
```

**Semantics:**
Class FamdictSet is used to represent a set of records selected from family dictionary table. It is inherited from class RecordSet.

- FamdictSet: class constructor
- ~FamdictSet: class destructor
- GetCountyName: It is used to get the county name of current record.
- GetCountyCode: It is used to get the county code of current record.

**Class DstrbSet: public CRecordSet**

```
public:
    DstrbSet();
    ~DstrbSet();
    CString GetPlant_id() { return Plant_id; }
    CString GetCounty_code() { return County_code; }
    CString GetSpecimen_year() { return Specimen_year; }
    CString GetSpecimen_date() { return Specimen_date; }
    CString GetSpecimen_id() { return Specimen_id; }
    CString GetCollector_name() { return Collector_name; }
    CString GetCollector_specimen_id()
    { return Collector_specimen_id; }
    CString GetSpecimen_locate() { return Specimen_locate; }
    CString GetSpecimen_development()
    {return Specimen_development; }
```
Object-oriented Design and Implementation of INVADERS System

```cpp
CString GetSpecimen_comments()
{  return Specimen_comments; }

CString GetData_enterer()
{  return Data_enterer; }

CTime GetEntry_date()
{  return Entry_date; }

protected:
  CString Plant_id;
  CString County_code;
  CString Specimen_year;
  CString Specimen_date;
  CString Specimen_id;
  CString Collector_name;
  CString Collector_specimen_id;
  CString Specimen_locate;
  CString Specimen_development;
  CString Specimen_comments;
  CString Data_enterer;
  CTime Entry_date;
};

Semantics:

Class DstrbSet is used to represent a set of records selected from distribution table. It is inherited from class RecordSet.

- DstrbSet: class constructor
- ~DstrbSet: class destructor
- GetPlant_id: return plant_id
- GetCounty_code: return county code
- GetSpecimen_year: return specimen year
- GetSpecimen_date: return specimen date
- GetSpecimen_id: return specimen_id
- GetCollector_name: return collector name
- GetCollector_specimen_id: return collector specimen_id
- GetSpecimen_locate: return specimen locate
- GetSpecimen_development: return specimen development
- GetSpecimen_comments: return specimen comments
- GetData_enterer: return data enterer
- GetEntry_date: return data entry date

Class MetaSet: public CRecordSet
{
  public:
    MetaSet();
}
~MetaSet();
CString GetFlora_id() { return Flora_id; }
CString GetSynonomy() { return Synonomy; }
CString GetPlant_id() { return Plant_id; }
CString GetGenus() { return Genus; }
CString GetSpecies() { return Species; }
CString GetAuthority() { return Authority; }
CString GetCommon_name() { return Common_name; }
CString GetOrigin() { return Origin; }
CString GetLifeform() { return Lifeform; }
CString GetStatus() { return Status; }
CString GetRegional_dstrb() { return Regional_dstrb; }
CString GetLife_zone() { return Life_zone; }
CString GetGlobal_dstrb() { return Global_dstrb; }
CString GetSub_level() { return Sub_level; }
CString GetSub_level_name() { return Sub_level_name; }
CString GetSub_level_authority() { return Sub_level_authority; }
CString GetData_enterer() { return Data_enterer; }
CTime GetEntry_date() { return Entry_date; }

protected:
CString Flora_id;
CString Synonomy;
CString Plant_id;
CString Genus;
CString Species;
CString Authority;
CString Common_name;
CString Origin;
CString Lifeform;
CString Status;
CString Regional_dstrb;
CString Life_zone;
CString Global_dstrb;
CString Sub_level;
CString Sub_level_name;
CString Sub_level_authority;
CString Data_enterer;
CTime Entry_date;

Semantics:
Class MetaSet is used to represent a set of records selected from meta

table. It is inherited from class RecordSet.
Object-oriented Design and Implementation of INVADERS System

- MetaSet: class constructor
- ~MetaSet: class destructor
- GetFlora_id: return flora_id
- GetSynonomy: return synonomy
- GetPlant_id: return plant_id
- GetGenus: return genus
- GetSpecies: return species
- GetAuthority: return authority
- GetCommon_name: return common name
- GetOrigin: return origin
- GetLifeform: return life form
- GetStatus: return status
- GetRegional_dstrb: return regional distribution
- GetLife_zone: return life zone
- GetGlobal_dstrb: return global_dstrb
- GetSub_level: return sub_level
- GetSub_level_name: return sub_level_name
- GetSub_level_authority: return sub_level_authority
- GetData_enterer: return data_enterer
- GetEntry_date: return data entry date

Class NoxlistSet: public CRecordSet
{
public:
    NoxlistSet();
    ~NoxlistSet();
    CString GetPlant_id() { return Plant_id; }  
    CString GetGenus() { return Genus; }  
    CString GetSpecies() { return Species; }  
    CString GetCommon_name() { return Common_name; }  
    CString GetFamily_name() { return Family_name; }  
    CString GetSub_level() { return Sub_level; }  
    CString GetSub_level_name() { return Sub_level_name; }  
    CString GetOrigin() { return Origin; }  
    CString GetLifeform() { return Lifeform; }  
    CString GetTNC_cat() { return TNC_cat; }  
    CString GetMT_cat() { return MT_cat; }  
    CString GetID_cat() { return ID_cat; }  
    CString GetOR_cat() { return OR_cat; }  
    CString GetWA_cat() { return WA_cat; }  
    CString GetWY_cat() { return WY_cat; }  
    CString GetFed_cat() { return Fed_cat; }  
    CString GetOther_cat() { return Other_cat; }  
}
protected:
    CString Plant_id;
    CString Genus;
    CString Species;
    CString Common_name;
    CString Family_name;
    CString Sub_level;
    CString Sub_level_name;
    CString Origin;
    CString Lifeform;
    CString TNC_cat;
    CString MT_cat;
    CString ID_cat;
    CString OR_cat;
    CString WA_cat;
    CString WY_cat;
    CString Fed_cat;
    CString Other_cat;
};

Semantics:
    Class NoxlistSet is used to represent a set of records selected from noxious plant table. It is inherited from class RecordSet.

    • NoxlistSet: class constructor
    • ~NoxlistSet: class destructor
    • GetPlant_id: return plant_id
    • GetGenus: return genus
    • GetSpecies: return species
    • GetCommon_name: return common name
    • GetFamily_name: return family name
    • GetSub_level: return sub_level
    • GetSub_level_name: return sub_level name
    • GetOrigin: return origin
    • GetLifeform: return life form
    • GetTNC_cat: return TNC_cat
    • GetMT_cat: return MT_cat
    • GetID_cat: return ID_cat
    • GetOR_cat: return OR_cat
    • GetWA_cat: return WA_cat
    • GetWY_cat: return WY_cat
    • GetFed_cat: return Fed_cat
    • GetOther_cat: return Other_cat
Class PlantSet: public CRecordSet
{
    public:
        PlantSet();
    ~PlantSet();
    CString GetPlant_id() { return Plant_id; }
    CString GetGenus() { return Genus; }
    CString GetSpecies() { return Species; }
    CString GetCommon_name() { return Common_name; }
    double GetDstrb_data() { return Dstrb_data; }
    CString GetEarliest_data() { return Earliest_data; }
    CString GetLatest_data() { return Latest_data; }

    protected:
        CString Plant_id;
        CString Genus;
        CString Species;
        CString Common_name;
        double Dstrb_data;
        CString Earliest_data;
        CString Latest_data;
};

Semantics:
    Class PlantSet is used to represent a set of records selected from plant table. It is inherited from class RecordSet.

- PlantSet: class constructor
- ~PlantSet: class destructor
- GetPlant_id: return plant_id
- GetGenus: return genus
- GetSpecies: return species
- GetCommon_name: return common name
- GetDstrb_data: return dstrb_data
- GetEarliest_data: return earliest data
- GetLatest_data: return latest_data

Class SensbSet: public CRecordSet
{
    public:
        SensbSet();
    ~SensbSet();
    CString GetPlant_id() { return Plant_id; }
    CString GetGenus() { return Genus; }
}
Object-oriented Design and Implementation of INVADERS System

```cpp
CString GetSpecies() { return Species; }
CString GetCommon_name() { return Common_name; }
CString GetFamily_name() { return Family_name; }
CString GetSub_level() { return Sub_level; }
CString GetSub_level_name() { return Sub_level_name; }
CString GetOrigin() { return Origin; }
CString GetLifeform() { return Lifeform; }
CString GetTNC_cat() { return TNC_cat; }
CString GetMT_cat() { return MT_cat; }
CString GetID_cat() { return ID_cat; }
CString GetOR_cat() { return OR_cat; }
CString GetWA_cat() { return WA_cat; }
CString GetWY_cat() { return WY_cat; }
CString GetUSFWS_cat() { return USFWS_cat; }
CString GetUSFS_cat() { return USFS_cat; }
CString GetFed_cat() { return Fed_cat; }
CString GetOther_cat() { return Other_cat; }
```

protected:
```
CString Plant_id;
CString Genus;
CString Species;
CString Common_name;
CString Family_name;
CString Sub_level;
CString Sub_level_name;
CString Origin;
CString Lifeform;
CString TNC_cat;
CString MT_cat;
CString ID_cat;
CString OR_cat;
CString WA_cat;
CString WY_cat;
CString USFWS_cat;
CString USFS_cat;
CString Fed_cat;
CString Other_cat;
```

Semantics:

Class SensbSet is used to represent a set of records selected from sensitive plant table. It is inherited from class RecordSet.

- SensbSet: class constructor
Object-oriented Design and Implementation of INVADERS System

- ~SensbSet: class destructor
- GetPlant_id: return plant_id
- GetGenus: return genus
- GetSpecies: return species
- GetCommon_name: return common name
- GetFamily_name: return family name
- GetSub_level: return sub_level
- GetSub_level_name: return sub_level name
- GetOrigin: return origin
- GetLifeform: return life form
- GetTNC_cat: return TNC_cat
- GetMT_cat: return MT_cat
- GetID_cat: return ID_cat
- GetOR_cat: return OR_cat
- GetWA_cat: return WA_cat
- GetWY_cat: return WY_cat
- GetUSFWS_cat: return USFWS_cat
- GetUSFS_cat: return USFS_cat
- GetFed_cat: return Fed_cat
- GetOther_cat: return Other_cat

Class SpidSet: public CRecordSet
{
    public:
        SpidSet();
        ~SpidSet();
        CString GetSpecimenId() { return Specimen_id; }
    protected:
        CString Specimen_id;
};

Semantics:
   Class SpidSet is used to represent a set of records selected from specimen_id table. It is inherited from class RecordSet.

- SpidSet: class constructor
- ~SpidSet: class destructor
- GetSpid: It is used to return the specimen ID of current record.

Class SrcSet: public CRecordSet
{
    public:

Object-oriented Design and Implementation of INVADERS System

```cpp
SrcSet();
~SrcSet();
CString GetSourceCode() { return Source_code; }
CString GetCodeDescription() { return Code_description; }
protected:
    CString Source_code;
    CString Code_description;
};

Semantics:
Class SrcSet is used to represent a set of records selected from src table. It is inherited from class RecordSet.

- SrcSet: class constructor
- ~SrcSet: class destructor
- GetSourceCode: It is used to return the source code of current record.
- GetCodeDescription: It is used to return the code description of current record.

Class Document::public CDocument
{
public:
    Document();
    ~Document();
    void OpenDocument();
    void CloseDocument();
protected:
    DstrbSet *dstrbSet;
    CountySet *countySet;
    FamdictSet *famdictSet;
    MetaSet *metaSet;
    PlantSet *plantSet;
    SensbSet *sensbSet;
    SpidSet *spidSet;
    SrcSet *srcSet;
    NoxlistSet *noxlistSet;
};

Semantics:
Class “Document” is used to contain and manage records set. The “View” class is attached to the document class and acts as an intermediary between the document and the user.
Object-oriented Design and Implementation of INVADERS System

- Document: class constructor
- ~Document: class destructor
- OpenDocument: This function is used to open a document.
- CloseDocument: This function is used to close a document.

**Class View::public CView**
{
    public:
    View();
    ~View();
    Document *GetDocument();
    void ClearScreen();
    void ComplexQuery();
    void OnPrint(CDC* pDC);
    void DisplayMap(CDC* pDC);
    void DisplayTable(CDC* pDC);
    void DisplayPlot(CDC* pDC);
}

Semantics:
Class View servers as a view window of our query result.

- View: class constructor
- ~View: class destructor
- GetDocument: It is used to get document pointer.
- ComplexQuery: It is used to query database by multiple query conditions.
- DisplayMap: It takes a pointer to CDC as argument and display distribution map.
- DisplayTable: It takes a pointer to CDC as argument and display records table
- DisplayPlot: It takes a pointer to CDC as argument and display plot of number of infested counties verses time lapse period.

**Class FrameWindow::public CFrameWnd**
{
    public:
    FrameWindow();
    ~FrameWindow();
    void CreateWindow();
    void DisplayTableButton();
    void DisplayMapButton();
    void DisplayPlotButton();

    protected:
CMenu wMenuBar;
CToolBar wToolBar;
CStatusBar wStatusBar;
};

Semantics:
Class Framewindow servers as a container of view. The frame window contains a menu bar, a tool bar, a status bar and a display area.

- FrameWindow: class constructor
- ~FrameWindow: class destructor
- CreateWindow: It is used to create frame window.
- DisplayTableButton: It is used to display table on the screen.
- DisplayMapButton: It is used to display distribution map on the screen.
- DisplayPlotButton: It is used to display plot on the screen.

Class QueryDlg::public CDialog
{
 public:
 QueryDlg();
 ~QueryDlg();
 InitialDialog();
 Press_OK_Button();
 Press_Cancel_Button();
 protected:
 CString state;
 CString county;
 CString latin_binomial;
 CString common_name;
 CString plant_id;
 List exotic_list;
 int query_stop_year;
};

Semantics:
Class QueryDlg is designed for query dialog. It supplies multiple query conditions to users and takes users query selection to INVADERS system.

- QueryDlg: class constructor
- ~QueryDlg: class destructor
- InitialDialog: initialize dialog
- On_OK_Button: It is used to implement actions when OK button is pushed.
• On_Cancel_Button: It is used to implement actions when Cancel button is pushed.

Class List::public Object
{
public:
    List();
~List();
    Bool LoadArchive(CFile filename, CArchive &ar);
    Bool StoreArchive(CFile filename, CArchive &ar);
protected:
    CArchive archive;
};

Semantics:
    Class list serves as a collection of a list of plant names.

• List: class constructor
• ~List: class destructor
• LoadArchive: This function is used to load data stored in a file to archive.
• StoreArchive: This function is used to store archive data to a file.

Class TimeLapseDlg::public CDialog
{
public:
    TimeLapseDlg();
~TimeLapseDlg();
    void InitialDialog();
    void Press_OK_Button();
    void Press_Cancel_Button();
protected:
    int timelapse;
};

Semantic:
    Class TimeLapseDlg is designed for “Time Lapse Dialog”. This dialog is used to take the value of time lapse period for display time lapse distribution map from user’s input.

• TimeLapseDlg: “Time Lapse Dialog” class constructor
• ~TimeLapseDlg: “Time Lapse Dialog” class destructor
• InitialDialog: This function is used to initialize “Time Lapse Dialog”.
• On_OK_Button: This function is used to implement actions when OK button is pushed.
• On_Cancel_Button: This function is used to implement actions when Cancel button is pushed.
Appendix B - Database Fields List

Meta table:

**Flora_id**---A unique eight-character (ch) code that is derived from a published name used for that plant.

**Synonymy code**---A six-ch code that specifies the accepted name for a plant.

**Plant_id**---An eight-ch code derived from the "accepted" name.

**Family_name**---A 20-ch field for the genus’ family name.

**Genus**---A 25-ch field for the genus name.

**Species**---A 28-ch field for the species name.

**Authority**---A 40-ch field for the naming authority.

**Common_name**---A 40-ch field for the species’ common name.

**Origin**---A one-ch code for the origin of the plant (E = exotic to North America, N = native to North America, U = unknown)

**Lifeform**---A one-ch code for the type of plant (F = forb, G = gram, L = lichen, M = moss, S = shrub, T = tree, E = fern).

**Status**---A one-ch code for indicating “concern” over the plant (S = sensitive, X noxious)

**Sub_level**---A 4-ch code for subdivision type.

**Sub_level_authority**---A 40-ch field for the naming authority of the subdivision.

Plant table:

**Plant_id**---Same as field in meta table.
**Genus**--- Same as field in meta table.

**Species**--- Same as field in meta table.

**Common_name**--- Same as field in meta table.

**Dstrb**--- A field storing the number of distribution records for this species in the dstrb table.

**Earliest**--- A date type field storing the date of the earliest distribution records for the species in the dstrb table.

**Latest**--- A date type field storing the date of the latest distribution records for the species in the dstrb table.

**Fam_dict table:**

**Genus**--- Same as field in meta table.

**Family_name**--- Same as field in meta table.

**Dstrb table:**

**Plant_id**--- Same as field in meta table.

**County_code**--- A five-character code for the state and county in which the specimen was collected. The first two letters are for the state, and the remaining letters are a three digit county code.

**Sp-date**--- A field that stores the date the specimen was collected.

**Sp-id**--- A fifteen-character code uniquely identifying the dstrb record. The first five letters are the herbarium source code and the remaining ten characters hold the herbarium's access code.
Collector---A thirty-character field for the specimen collector's name.

Sp-locale---A twenty-character field for a specific locale where the specimen was found.

Sp-develop---A ten-character field for the development stage of the specimen (fruit, flower, veg)

Sp-comments---A hundred-character field for any comments about the specimen.

County table:

County_name---A fifteen-character field stores county name.

County_code---A five-character code stores county code.

Noxlist table:

Plant_id---Same as field in meta table.

Genus---Same as field in meta table.

Species---Same as field in meta table.

Common_name---Same as field in meta table.

Family_name---Same as field in meta table.

Sub_level---Same as field in meta table.

Sub_level_name---Same as field in meta table.

Origin---Same as field in meta table.

Lifeform---Same as field in meta table.

Global_cat---not used
**ID_cat**---indicate whether this plant is listed as a noxious plant in state Idaho

**MT_cat**--- indicate whether this plant is listed as a noxious plant in state Montana

**WY_cat**--- indicate whether this plant is listed as a noxious plant in state Wyoming

**WA_cat**--- indicate whether this plant is listed as a noxious plant by state Washington

**OR_cat**--- indicate whether this plant is listed as a noxious plant by state Oregon

**USFWS_cat**---not used

**USFS_cat**---not used

**Fed_cat**--- indicate whether this plant is listed as a noxious plant by federal government

**Other_cat**--- indicate whether this plant is listed as a noxious plant by other source

**Sensb table:** It has the same structure as Noxlist table. Fields Global_cat, USFWS_cat and USFS_cat are used for indicating whether this plant is listed as a sensitive (rare, endangered or threatened) native plant by these source.
Reference


