**INDEX**

1. **Contents 5** 
   1. Introduction
   2. System Analysis

1.2.1 Existing system

1.2.2 Proposed system

1. **Module 9**
   1. Module Description
   2. System features
2. **Physical Design 12**
   1. Data Flow Diagrams
   2. Data Dictionary
   3. Entity Relationship Diagram
   4. Requirements
      1. Hardware Requirements
      2. Software Requirements
   5. Use-Case Diagrams
   6. Class Diagrams
   7. Activity Diagram
3. **About JAVA 23**
   1. Features of Java
   2. JDBC
   3. JSP
4. **About ORACLE 38** 
   1. DBMS
   2. RDBMS
   3. SQL
5. **Testing 47**
6. **Screens**
7. **Conclusion**
8. **References**

**1. CONTENTS**

**1.1 INTRODUCTION:**

Vehicle Monitoring System is used to track the data of the vehicle and about the new vehicles if added to particular unit or section. This module also captures the information on vehicle problems, their repair details and vehicle transfer details. All the details regarding vehicle monitoring are maintained in the database. Following are the activities performed during the process.

**1.2 SYSTEM ANALYSIS:**

**Problem definition:**

**Proposed System:**

It can be to ensure that everything gets done.  Each and every vehicle and piece of equipment needs to receive the right kind of service at the right time.  In addition, you need to keep track of unexpected repairs and problems so you can accurately determine the cost effectiveness of your equipment.

**1.2.1 Existing system:**

The Vehicle Monitoring system can show all kinds of information, including the position of vehicles at any time, the time taken to complete journeys and how long the vehicle spent at a particular location. The technology may be cutting edge, but the idea is simple -our monitoring solution means that you can get the best from your vehicles 24-hours a day. Vehicle Monitoring is about increasing your productivity and profitability. Vehicle Monitoring brings you a well designed, efficient, cost effective vehicle management solution

**2. MODULE**

The modules present in the online crime reporting are:

**MODULES:**

* **Admin**
* **Distributor**
* **User**

**MODULE DESCRIPTION:**

* **User Module:**

**Vehicle Problem Reporting**

If there are any problems relating to the vehicle they are recorded solved. Seriousness of the problem is recorded into the database.

**Vehicle Requirements**

Departments utilize vehicle to carry out their activities. A system for vehicle monitoring and tracking vehicle requirements should be maintained. Details regarding vehicle requirements are to be captured and stored in the database.

**Distributor**:

**Vehicle Distribution**

New vehicles are distributed department wise and allotted to the units whoever in need of vehicle in the unit or department can use it as per requirement.

**Vehicle Transfers**

Vehicles can be transferred from one unit to another and one department to another. All these details are stored into the database. Whenever any vehicle is transferred from one unit to unit the details are manipulated in the database. Also if it is transferred from one official to another the details are stored into the database

**Admin**

**New vehicles**

Whenever new vehicles are received respective department of health should be intimated. Also the database should be updated regularly whenever a new vehicle is received. Also the department intimates whenever it receives new vehicles to carry out its operations successfully

**Vehicle usage details**

Details regarding the movement of vehicles are captured. It gathers the detailed usage of the trip vehicle has made. It also captures the number of cases shifted using the vehicle.

The process facilitates to view the usage details of a vehicle. Even the reports should be generated monthly to know the number of cases delivered in that vehicle.

Create a form to add, modify trip details. Users are allowed to view and delete the vehicle usage details.

**Microsoft.NET Framework**

The .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet. The .NET Framework is designed to fulfill the following objectives:

To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.

To provide a code-execution environment that minimizes software deployment and versioning conflicts.

To provide a code-execution environment that guarantees safe execution of code, including code created by an unknown or semi-trusted third party.

To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.

To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.

To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework. You can think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that ensure security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

The .NET Framework can be hosted by unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software environment that can exploit both managed and unmanaged features. The .NET Framework not only provides several runtime hosts, but also supports the development of third-party runtime hosts.

For example, ASP.NET hosts the runtime to provide a scalable, server-side environment for managed code. ASP.NET works directly with the runtime to enable Web Forms applications and XML Web services, both of which are discussed later in this topic.

Internet Explorer is an example of an unmanaged application that hosts the runtime (in the form of a MIME type extension). Using Internet Explorer to host the runtime enables you to embed managed components or Windows Forms controls in HTML documents. Hosting the runtime in this way makes managed mobile code (similar to Microsoft® ActiveX® controls) possible, but with significant improvements that only managed code can offer, such as semi-trusted execution and secure isolated file storage.

The following illustration shows the relationship of the common language runtime and the class library to your applications and to the overall system. The illustration also shows how managed code operates within a larger architecture.

**Features of the Common Language Runtime**

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on the common language runtime.

With regards to security, managed components are awarded varying degrees of trust, depending on a number of factors that include their origin (such as the Internet, enterprise network, or local computer). This means that a managed component might or might not be able to perform file-access operations, registry-access operations, or other sensitive functions, even if it is being used in the same active application.

The runtime enforces code access security. For example, users can trust that an executable embedded in a Web page can play an animation on screen or sing a song, but cannot access their personal data, file system, or network. The security features of the runtime thus enable legitimate Internet-deployed software to be exceptionally feature rich.

The runtime also enforces code robustness by implementing a strict type- and code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing. The various Microsoft and third-party language compilers

Generate managed code that conforms to the CTS. This means that managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

In addition, the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common application errors, memory leaks and invalid memory references.

The runtime also accelerates developer productivity. For example, programmers can write applications in their development language of choice, yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing applications.

While the runtime is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary COM components and DLLs.

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language of the system on which it is executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance.

Finally, the runtime can be hosted by high-performance, server-side applications, such as Microsoft® SQL Server™ and Internet Information Services (IIS). This infrastructure enables you to use managed code to write your business logic, while still enjoying the superior performance of the industry's best enterprise servers that support runtime hosting.

**.NET Framework Class Library**

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime. The class library is object oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new

features of the .NET Framework. In addition, third-party components can integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces that you can use to develop your own collection classes. Your collection classes will blend seamlessly with the classes in the .NET Framework.

As you would expect from an object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. For example, you can use the .NET Framework to develop the following types of applications and services:

Console applications.

Scripted or hosted applications.

Windows GUI applications (Windows Forms).

ASP.NET applications.

XML Web services.

Windows services.

For example, the Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development. If you write an ASP.NET Web Form application, you can use the Web Forms classes.

**Client Application Development**

Client applications are the closest to a traditional style of application in Windows-based programming. These are the types of applications that display windows or forms on the desktop, enabling a user to perform a task. Client applications include applications such as word processors and spreadsheets, as well as custom business applications such as data-entry tools, reporting tools, and so on. Client applications usually employ windows, menus, buttons, and other GUI elements, and they likely access local resources such as the file system and peripherals such as printers.

Another kind of client application is the traditional ActiveX control (now replaced by the managed Windows Forms control) deployed over the Internet as a Web page. This application is much like other client applications: it is executed natively, has access to local resources, and includes graphical elements.

In the past, developers created such applications using C/C++ in conjunction with the Microsoft Foundation Classes (MFC) or with a rapid application development (RAD) environment such as Microsoft® Visual Basic®. The .NET Framework incorporates aspects of these existing products into a single, consistent development environment that drastically simplifies the development of client applications.

The Windows Forms classes contained in the .NET Framework are designed to be used for GUI development. You can easily create command windows, buttons, menus, toolbars, and other screen elements with the flexibility necessary to accommodate shifting business needs.

For example, the .NET Framework provides simple properties to adjust visual attributes associated with forms. In some cases the underlying operating system does not support changing these attributes directly, and in these cases the .NET Framework automatically recreates the forms. This is one of many ways in which the .NET Framework integrates the developer interface, making coding simpler and more consistent.

Unlike ActiveX controls, Windows Forms controls have semi-trusted access to a user's computer. This means that binary or natively executing code can access some of the resources on the user's system (such as GUI elements and limited file access) without being able to access or compromise other resources. Because of code access security, many applications that once needed to be installed on a user's system can now be safely deployed through the Web. Your applications can implement the features of a local application while being deployed like a Web page.

C#.Net for Windows Application

**Overview of the .NET Framework**

The .NET Framework is a managed type-safe environment for application development and execution. The .NET Framework manages all aspects of your program’s execution. It allocates memory for the storage of data and instructions, grants or denies the appropriate permissions to your application, initiates and manages application execution, and manages the reallocation of memory from resources that are no longer needed. The .NET Framework consists of two main components: the common language runtime and the .NET Framework class library.

The common language runtime can be thought of as the environment that manages code execution. It provides core services, such as code compilation, memory allocation, thread management, and garbage collection. Through the common type system (CTS), it enforces strict type-safety and ensures that code is executed in a safe environment by also enforcing code access security.

The .NET Framework class library provides a collection of useful and reusable types that are designed to integrate with the common language runtime. The types provided by the .NET Framework are object-oriented and fully extensible, and they allow you to seamlessly integrate your applications with the .NET Framework.

**Languages and the .NET Framework**

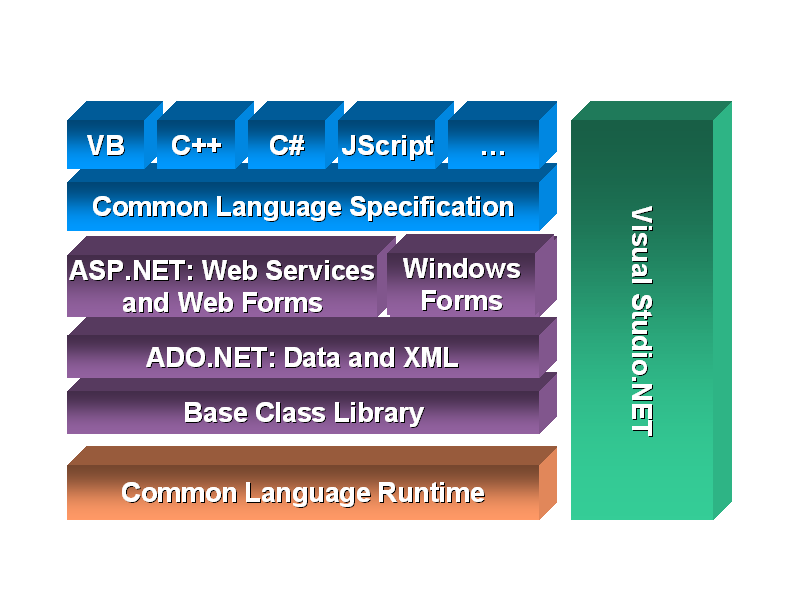
The .NET Framework is designed for cross-language compatibility, which means, simply, that .NET components can interact with each other no matter what supported language they were written in originally. So, an application written in Microsoft Visual Basic .NET might reference a dynamic-link library (DLL) file written in Microsoft Visual C#, which in turn might access a resource written in managed Microsoft Visual C++ or any other .NET language. This language interoperability extends to full object-oriented inheritance. A Visual Basic .NET class might be derived from a C# class, for example, or vice versa.

This level of cross-language compatibility is possible because of the common language runtime. When a .NET application is compiled, it is converted from the language in which it was written (Visual Basic .NET, C#, or any other .NET-compliant language) to Microsoft Intermediate Language (MSIL or IL). MSIL is a low-level language that the common language runtime can read and understand. Because all .NET executables and DLLs exist as MSIL, they can freely interoperate. The Common Language Specification (CLS) defines the minimum standards to which .NET language compilers must conform. Thus, the CLS ensures that any source code successfully compiled by a .NET compiler can interoperate with the .NET Framework.

The CTS ensures type compatibility between .NET components. Because .NET applications are converted to IL prior to deployment and execution, all primitive data types are represented as .NET types. Thus, a Visual Basic Integer and a C# int are both represented in IL code as a System.Int32. Because both languages use a common type system, it is possible to transfer data between components and avoid time-consuming conversions or hard-to-find errors.

Visual Studio .NET ships with languages such as Visual Basic .NET, Visual C#, and Visual C++ with managed extensions, as well as the JScript scripting language. You can also write managed code for the .NET Framework in other languages. Third-party tools and compilers exist for Fortran, Cobol, Perl, and a host of other languages. All of these languages share the same cross-language compatibility and inheritability. Thus, you can write code for the .NET Framework in the language of your choice, and it will be able to interact with code written for the .NET Framework in any other language.

.**NET Framework Architecture**



The Structure of a .NET Application

To understand how the common language runtime manages code execution, you must examine the structure of a .NET application. The primary unit of a .NET application is the assembly. An assembly is a self-describing collection of code, resources, and metadata. The assembly manifest contains information about what is contained within the assembly. The assembly manifest provides:

Identity information, such as the assembly’s name and version number

A list of all types exposed by the assembly

A list of other assemblies required by the assembly

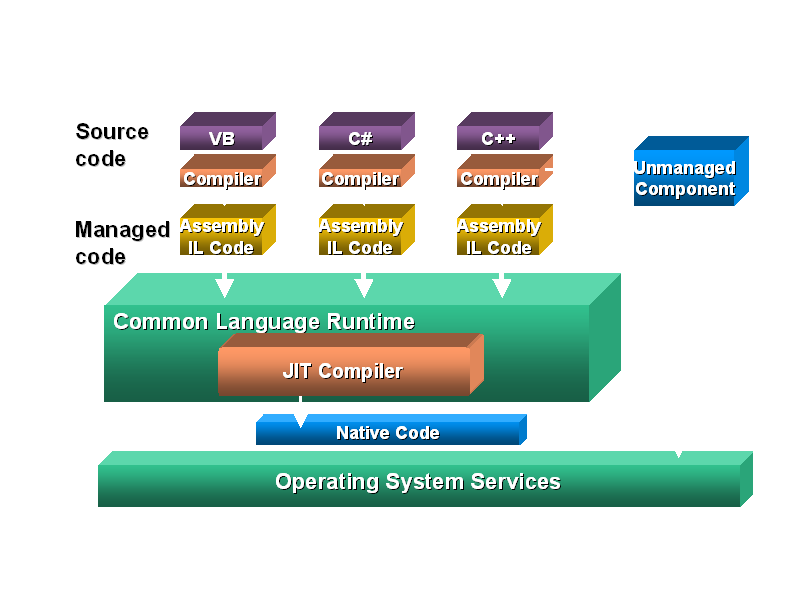
A list of code access security instructions, including permissions required by the assembly and permissions to be denied the assembly

Each assembly has one and only one assembly manifest, and it contains all the description information for the assembly. However, the assembly manifest can be contained in its own file or within one of the assembly’s modules.

An assembly contains one or more modules. A module contains the code that makes up your application or library, and it contains metadata that describes that code. When you compile a project into an assembly, your code is converted from high-level code to IL. Because all managed code is first converted to IL code, applications written in different languages can easily interact. For example, one developer might write an application in Visual C# that accesses a DLL in Visual Basic .NET. Both resources will be converted to IL modules before being executed, thus avoiding any language-incompatibility issues.

Each module also contains a number of types. Types are templates that describe a set of data encapsulation and functionality. There are two kinds of types: reference types (classes) and value types (structures). These types are discussed in greater detail in Lesson 2 of this chapter. Each type is described to the common language runtime in the assembly manifest. A type can contain fields, properties, and methods, each of which should be related to a common functionality. For example, you might have a class that represents a bank account. It contains fields, properties, and methods related to the functions needed to implement a bank account. A field represents storage of a particular type of data. One field might store the name of an account holder, for example. Properties are similar to fields, but properties usually provide some kind of validation when data is set or retrieved. You might have a property that represents an account balance. When an attempt is made to change the value, the property can check to see if the attempted change is greater than a predetermined limit. If the value is greater than the limit, the property does not allow the change. Methods represent behavior, such as actions taken on data stored within the class or changes to the user interface. Continuing with the bank account example, you might have a Transfer method that transfers a balance from a checking account to a savings account, or an Alert method that warns users when their balances fall below a predetermined level.

**CLR Execution Model**



Compilation and Execution of a .NET Application

When you compile a .NET application, it is not compiled to binary machine code; rather, it is converted to IL. This is the form that your deployed application takes—one or more assemblies consisting of executable files and DLL files in IL form. At least one of these assemblies will contain an executable file that has been designated as the entry point for the application.

When execution of your program begins, the first assembly is loaded into memory. At this point, the common language runtime examines the assembly manifest and determines the requirements to run the program. It examines security permissions requested by the assembly and compares them with the system’s security policy. If the system’s security policy does not allow the requested permissions, the application will not run. If the application passes the system’s security policy, the common ­language runtime executes the code. It creates a process for the application to run in and begins application execution. When execution starts, the first bit of code that needs to be executed is loaded into memory and compiled into native binary code from IL by the common language runtime’s Just-In-Time (JIT) compiler. Once compiled, the code is executed and stored in memory as native code. Thus, each portion of code is compiled only once when an application executes. Whenever program execution branches to code that has not yet run, the JIT compiler compiles it ahead of execution and stores it in memory as binary code. This way, application performance is maximized because only the parts of a program that are executed are compiled.

**2:** **The .NET Base Class Library**

The .NET base class library is a collection of object-oriented types and interfaces that provide object models and services for many of the complex programming tasks you will face. Most of the types presented by the .NET base class library are fully extensible, allowing you to build types that incorporate your own functionality into your managed code.

The .NET Framework base class library contains the base classes that provide many of the services and objects you need when writing your applications. The class library is organized into namespaces. A namespace is a logical grouping of types that perform related functions. For example, the System.Windows.Forms namespace contains all the types that make up Windows forms and the controls used in those forms.

Namespaces are logical groupings of related classes. The namespaces in the .NET base class library are organized hierarchically. The root of the .NET Framework is the System namespace. Other namespaces can be accessed with the period operator. A typical namespace construction appears as follows:

System

System. Data

System.Data.SQLClient

The first example refers to the System namespace. The second refers to the System.Data namespace. The third example refers to the System.Data.SQLClient namespace. Table 1.1 introduces some of the more commonly used .NET base class namespaces.

| Table 1-1. Representative .NET Namespaces | |
| --- | --- |
| Namespace | Description |
| System | This namespace is the root for many of the low-level types required by the .NET Framework. It is the root for primitive data types as well, and it is the root for all the other namespaces in the .NET base class library. |
| System.Collections | This namespace contains classes that represent a variety of different container types, such as ArrayList, SortedList, Queue, and Stack. You also can find abstract classes, such as CollectionBase, which are useful for implementing your own collection functionality. |
| System.ComponentModel | This namespace contains classes involved in component creation and containment, such as attributes, type converters, and license providers. |
| System.Data | This namespace contains classes required for database access and manipulations, as well as additional namespaces used for data access. |
| System.Data.Common | This namespace contains a set of classes that are shared by the .NET managed data providers. |
| System.Data.OleDb | This namespace contains classes that make up the managed data provider for OLE DB data access. |
| System.Data.SQLClient | This namespace contains classes that are optimized for interacting with Microsoft SQL Server. |
| System.Drawing | This namespace exposes GDI+ functionality and provides classes that facilitate graphics rendering. |
| System.IO | In this namespace, you will find types for handling file system I/O. |
| System.Math | This namespace is home to common mathematics functions such as extracting roots and trigonometry. |
| System.Reflection | This namespace provides support for obtaining information and dynamic creation of types at runtime. |
| System.Security | This namespace is home to types dealing with permissions, cryptography, and code access security. |
| System.Threading | This namespace contains classes that facilitate the implementation of multithreaded applications. |
| System.Windows.Forms | This namespace contains types involved in creating standard Windows applications. Classes that represent forms and controls reside here as well. |

The namespace names are self-descriptive by design. Straightforward names make the .NET Framework easy to use and allow you to rapidly familiarize yourself with its contents.

Reference Types and Value Types

Types in the .NET Framework come in two varieties: value types and reference types. The primary difference between value types and reference types has to do with the way variable data is accessed. To understand this difference, a little background on memory dynamics is required.

Application data memory is divided into two primary components, the stack and the heap. The stack is an area of memory reserved by the application to run the program. The stack is analogous to a stack of dinner plates. Plates are placed on the stack one on top of another. When a plate is removed from the stack, it is always the last one to have been placed on top that is removed first. So it is with program variables. When a function is called, all the variables used by the function are pushed onto the stack. If that function calls additional functions, it pushes additional variables onto the stack. When the most recently called function terminates, all of its variables go out of scope (meaning that they are no longer available to the application) and are popped off the stack. Memory consumed by those variables is then freed up, and program execution continues.

The heap, on the other hand, is a separate area of memory reserved for the creation of reusable objects. The common language runtime manages allocation of heap memory for objects and controls the reclamation of memory from unused objects through garbage collection.

All the data associated with a value type is allocated on the stack. When a variable of a value type goes out of scope, it is destroyed and its memory is reclaimed. A variable of a reference type, on the other hand, exists in two memory locations. The actual object data is allocated on the heap. A variable containing a pointer to that object is allocated on the stack. When that variable is called by a function, it returns the memory address for the object to which it refers. When that variable goes out of scope, the object reference is destroyed but the object itself is not. If any other references to that object exist, the object remains intact. If the object is left without any references, it is subject to garbage collection. (See Lesson 6 of this chapter.)

Examples of value types include primitives, such as Integer (int), Boolean (bool), Char (char), and so on, as well as user-defined types such as Structure (struct) and Enumeration (enum). Classes represent the majority of reference types. Other reference types include the interface, delegate, and array types. Classes and structures are discussed in Lesson 3 of this chapter, and other reference and value types are discussed in Chapter 3.

Using .NET Framework Types in Your Application

When you begin writing an application, you automatically begin with a reference to the .NET Framework base class library. You reference it so that your application is aware of the base class library and is able to create instances of the types represented by it.

Value Types

int myInteger;

This line tells the runtime to allocate the appropriate amount of memory to hold an integer variable. Although this line creates the variable, it does not assign a value to it. You can assign a value using the assignment operator, as follows:

myInteger = 42;

You can also choose to assign a value to a variable upon creation, as shown in this example:

int myInteger = 42;

Reference Types

Creating an instance of a type is a two-step process. The first step is to declare the variable as that type, which allocates the appropriate amount of memory for that variable but does not actually create the object. The following syntax declares an object:

System.Windows.Forms.Form myForm;

This line tells the runtime to set aside enough memory to hold a Form variable and assigns it the name myForm, but it does not actually create the Form object in memory. The second step, called instantiation, actually creates the object. An example of instantiation follows:

myForm = new System.Windows.Forms.Form();

This line makes a call to the constructor method of the type System.Windows.Forms.Form by way of the New (new) keyword. The constructor is a special method that is invoked only at the beginning of an object’s lifetime. It contains any code that must be executed for the object to work (assigning values to properties, for example). If any parameters were required by the constructor, they would be contained within the parentheses at the end of the line. The following example shows declaration and instantiation of a hypothetical Widget class that requires a string as a parameter in the constructor.

Widget myWidget;

myWidget = new Widget("This string is required by the constructor");

If desired, you can also combine both declaration and instantiation into a single statement. By declaring and instantiating an object in the same line, you reserve the memory for the object and immediately create the object that resides in that memory. Although there was a significant performance penalty for this shortcut in previous versions of Visual Basic, Visual Basic .NET and Visual C# are optimized to allow this behavior without any performance loss. The following example shows the one-step declaration and instantiation of a new Form:

System.Windows.Forms.Form myForm = new

System.Windows.Forms.Form();

Both value types and reference types must be initialized before use. For class and structure fields in Visual Basic .NET, types are initialized with default values on declaration. Numeric value types (such as integer) and floating-point types are assigned zero; Boolean variables are assigned False; and reference types are assigned to a null reference.

In C#, variables of a reference type have a default value of null. It is recommended that you do not rely on the default value. These variables should not be used until they have been initialized.

Using Value Type and Reference Type Variables

A variable that represents a value type contains all the data represented by that type. A variable that represents a reference type contains a reference to a particular object. This distinction is important. Consider the following example:

int x, y;

x = 15;

y = x;

x = 30;

// What is the value of y?

In this example, two integer variables named x and y are created. X is assigned a value of 15, and then y is assigned the value of x. Next the value of x is changed to 30, and the question is posed: what is the value of y? The answer to this question might seem obvious, and it is y = 15 because x and y are two separate variables and have no effect on each other when changed. When the line y = x is encountered, the value of x is copied to the value of y, and there is no further connection between the two variables.

This situation changes, however, in the case of reference types. Let’s reconsider the previous example using a reference type (Form) instead of a value type.

System.Windows.Forms.Form x,y;

x = new System.Windows.Forms.Form();

x.Text = "This is Form 1";

y = x;

x.Text = "This is Form 2";

// What value does y.Text return?

What value does y.Text return? This time, the answer is less obvious. Because System.Windows.Forms.Form is a reference type, the variable x does not actually contain a Form; rather, it points to an instance of a Form. When the line y = x is encountered, the runtime copies the reference from variable x to y. Thus, the variables x and y now point to the same instance of Form. Because these two variables refer to the same instance of the object, they will return the same values for properties of that object. Thus, y.Text returns “This is Form 2”.

The Imports and Using Statements

Up to this point of the chapter, if you wanted to access a type in the .NET Framework base class library, you had to use the full name of the type, including every namespace to which it belonged. For example:

System.Windows.Forms.Form

This is called the fully-qualified name, meaning it refers both to the class and to the namespace in which it can be found. You can make your development environment “aware” of various namespaces by using the Imports (Visual Basic .NET) or using (Visual C#) statement. This technique allows you to refer to a type using only its generic name and to omit the qualifying namespaces. Thus, you could refer to System.Windows.Forms.Form as simply Form. In Visual Basic .NET, the Imports statement must be placed at the top of the code window, preceding any other statement (except Option). In Visual C#, the using statement must occur before any other namespace element, such as a class or struct. This example demonstrates use of this statement:

Using System.Windows.Forms;

When two types of the same name exist in more than one imported namespace, you must use the fully qualified name to avoid a naming conflict. Thus, if you are using MyNameSpaceOne and MyNameSpaceTwo, and each contains a Widget class, you would have to refer to MyNameSpaceOne.Widget or MyNameSpaceTwo.Widget to ensure the correct result.

In C#, you can resolve namespace conflicts such as these by creating an alias. An alias allows you to choose one name to refer to another class. You create an alias using the using keyword, as shown below:

Using myAlias = MyNameSpaceTwo.Widget;

After implementing an alias, you can use it in code to represent the aliased class. For example:

// You can now refer to MyNameSpaceTwo as myAlias. The

// following two lines produce the same result:

MyNameSpaceTwo.Widget anotherWidget = new MyNameSpaceTwo.Widget() ;

myAlias another Widget = new mYAlias() ;

You cannot create aliases for types in this manner in Visual Basic .NET.

Referencing External Libraries

You might want to use class libraries not contained by the .NET Framework, such as libraries developed by third-party vendors or libraries you developed. To access these external libraries, you must create a reference.

To create a reference to an external library

In the Solution Explorer, right-click the References node of your project.

From the pop-up menu, choose Add Reference. The Add Reference dialog box appears.

Choose the appropriate tab for the library you want to reference. .NET libraries are available on the .NET tab. Legacy COM libraries appear on the COM tab, and local Visual Studio projects appear on the Projects tab.

Locate the library you want to reference, and double-click it to add it to the Selected components box. Click OK to confirm the choice of that reference.

**Introduction to Object-Oriented Programming**

Programming in the .NET Framework environment is done with objects. Objects are programmatic constructs that represent packages of related data and functionality. Objects are self-contained and expose specific functionality to the rest of the application environment without detailing the inner workings of the object itself. Objects are created from a template called a class. The .NET base class library provides a set of classes from which you can create objects in your applications. You also can use the Microsoft Visual Studio programming environment to create your own classes. This lesson introduces you to the concepts associated with object-oriented programming.

**Objects, Members, and Abstraction**

An object is a programmatic construct that represents something. In the real world, objects are cars, bicycles, laptop computers, and so on. Each of these items exposes specific functionality and has specific properties. In your application, an object might be a form, a control such as a button, a database connection, or any of a number of other constructs. Each object is a complete functional unit, and contains all of the data and exposes all of the functionality required to fulfill its purpose. The ability of programmatic objects to represent real-world objects is called abstraction.

**Classes Are Templates for Objects**

Classes can be thought of as blueprints for objects: they define all of the members of an object, define the behavior of an object, and set initial values for data when appropriate. When a class is instantiated, an in-memory instance of that class is created. This instance is called an object. To review, a class is instantiated using the New (new) keyword as follows:

When an instance of a class is created, a copy of the instance data defined by that class is created in memory and assigned to the reference variable. Individual instances of a class are independent of one another and represent separate programmatic constructs. There is generally no limit to how many copies of a single class can be instantiated at any time. To use a real-world analogy, if a car is an object, the plans for the car are the class. The plans can be used to make any number of cars, and changes to a single car do not, for the most part, affect any other cars.

**Objects and Members**

Objects are composed of members. Members are properties, fields, methods, and events, and they represent the data and functionality that comprise the object. Fields and properties represent data members of an object. Methods are actions the object can perform, and events are notifications an object receives from or sends to other objects when activity happens in the application.

To continue with the real-world example of a car, consider that a Car object has fields and properties, such as Color, Make, Model, Age, GasLevel, and so on. These are the data that describe the state of the object. A Car object might also expose several methods, such as Accelerate, ShiftGears, or Turn. The methods represent behaviors the object can execute. And events represent notifications. For example, a Car object might receive an EngineOverheating event from its Engine object, or it might raise a Crash event when interacting with a Tree object.

**Object Models**

Simple objects might consist of only a few properties, methods, and perhaps an event or two. More complex objects might require numerous properties and methods and possibly even subordinate objects. Objects can contain and expose other objects as members. For example, the TextBox control exposes a Font property, which consists of a Font object. Similarly, every instance of the Form class contains and exposes a Controls collection that comprises all of the controls contained by the form. The object model defines the hierarchy of contained objects that form the structure of an object.

An object model is a hierarchical organization of subordinate objects contained and exposed within a main object. To illustrate, let’s revisit the example of a car as an object. A car is a single object, but it also consists of subordinate objects. A Car object might contain an Engine object, four Wheel objects, a Transmission object, and so on. The composition of these subordinate objects directly affects how the Car object functions as a whole. For example, if the Cylinders property of the Engine subordinate object is equal to 4, the Car will behave differently than a Car whose Engine has a Cylinders property value of 8. Contained objects can have subordinate objects of their own. For example, the contained Engine object might contain several SparkPlug objects.

**Encapsulation**

Encapsulation is the concept that implementation of an object is independent of its interface. Put another way, an application interacts with an object through its interface, which consists of its public properties and methods. As long as this interface remains constant, the application can continue to interact with the component, even if implementation of the interface was completely rewritten between versions.

Objects should only interact with other objects through their public methods and properties. Thus, objects should contain all of the data they require, as well as all of the functionality that works with that data. The internal data of an object should never be exposed in the interface; thus, fields rarely should be Public (public).

Returning to the Car example. If a Car object interacts with a Driver object, the Car interface might consist of a GoForward method, a GoBackward method, and a Stop method. This is all the information that the Driver needs to interact with the Car. The Car might contain an Engine object, for example, but the Driver doesn’t need to know about the Engine object—all the Driver cares about is that the methods can be called and that they return the appropriate values. Thus, if one Engine object is exchanged for another, it makes no difference to the Driver as long as the interface continues to function correctly.

**Polymorphism**

Polymorphism is the ability of different classes to provide different implementations of the same public interfaces. In other words, polymorphism allows methods and properties of an object to be called without regard for the particular implementation of those members. For example, a Driver object can interact with a Car object through the Car public interface. If another object, such as a Truck object or a SportsCar object, exposes the same public interface, the Driver object can interact with them without regard to the specific implementation of that interface. There are two principal ways through which polymorphism can be provided: interface polymorphism and inheritance polymorphism.

**Interface Polymorphism**

An interface is a contract for behavior. Essentially, it defines the members a class should implement, but states nothing at all about the details of that implementation. An object can implement many different interfaces, and many diverse classes can implement the same interface. All objects implementing the same interface are capable of interacting with other objects through that interface. For example, the Car object in the previous examples might implement the IDrivable interface (by convention, interfaces usually begin with I), which specifies the GoForward, GoBackward, and Halt methods. Other classes, such as Truck, Forklift, or Boat might implement this interface and thus are able to interact with the Driver object. The Driver object is unaware of which interface implementation it is interacting with; it is only aware of the interface itself. Interface polymorphism is discussed in detail in Lesson 3.

**Inheritance Polymorphism**

Inheritance allows you to incorporate the functionality of a previously defined class into a new class and implement different members as needed. A class that inherits another class is said to derive from that class, or to inherit from that class. A class can directly inherit from only one class, which is called the base class. The new class has the same members as the base class, and additional members can be added as needed. Additionally, the implementation of base members can be changed in the new class by overriding the base class implementation. Inherited classes retain all the characteristics of the base class and can interact with other objects as though they were instances of the base class. For example, if the Car class is the base class, a derived class might be SportsCar. The SportsCar class might be the base class for another derived class, the ConvertibleSportsCar. Each newly derived class might implement additional members, but the functionality defined in the original Car class is retained.

**3. PHYSICAL DESIGN**

**3.1. DATA FLOW DIAGRAMS**

A data flow diagram is graphical tool used to describe and analyze movement of data through a system. These are the central tool and the basis from which the other components are developed. The transformation of data from input to output, through processed, may be described logically and independently of physical components associated with the system. These are known as the logical data flow diagrams. The physical data flow diagrams show the actual implements and movement of data between people, departments and workstations. A full description of a system actually consists of a set of data flow diagrams. Using two familiar notations Yourdon, Gane and Sarson notation develops the data flow diagrams. Each component in a DFD is labeled with a descriptive name. Process is further identified with a number that will be used for identification purpose. The development of DFD’s is done in several levels. Each process in lower level diagrams can be broken down into a more detailed DFD in the next level. The lop-level diagram is often called context diagram. It consists a single process bit, which plays vital role in studying the current system. The process in the context level diagram is exploded into other process at the first level DFD.

The idea behind the explosion of a process into more process is that understanding at one level of detail is exploded into greater detail at the next level. This is done until further explosion is necessary and an adequate amount of detail is described for analyst to understand the process.

Larry Constantine first developed the DFD as a way of expressing system requirements in a graphical from, this lead to the modular design.

A DFD is also known as a “bubble Chart” has the purpose of clarifying system requirements and identifying major transformations that will become programs in system design. So it is the starting point of the design to the lowest level of detail. A DFD consists of a series of bubbles joined by data flows in the system.

**DFD SYMBOLS:**In the DFD, there are four symbols

A square defines a source (originator) or destination of system data

An arrow identifies data flow. It is the pipeline through which the information flows

A circle or a bubble represents a process that transforms incoming data flow into outgoing data flows.

An open rectangle is a data store, data at rest or a temporary repository of data

Process that transforms data flow.

Source or Destination of data

Data flow

Data Store

CONSTRUCTING A DFD:

Several rules of thumb are used in drawing DFD’s:

Process should be named and numbered for an easy reference. Each name should be representative of the process.

The direction of flow is from top to bottom and from left to right. Data Traditionally flow from source to the destination although they may flow back to the source. One way to indicate this is to draw long flow line back to a source. An alternative way is to repeat the source symbol as a destination. Since it is used more than once in the DFD it is marked with a short diagonal.

When a process is exploded into lower level details, they are numbered.

The names of data stores and destinations are written in capital letters. Process and dataflow names have the first letter of each work capitalized

A DFD typically shows the minimum contents of data store. Each data store should contain all the data elements that flow in and out.

Questionnaires should contain all the data elements that flow in and out. Missing interfaces redundancies and like is then accounted for often through interviews.

SAILENT FEATURES OF DFD’s

The DFD shows flow of data, not of control loops and decision are controlled considerations do not appear on a DFD.

The DFD does not indicate the time factor involved in any process whether the dataflows take place daily, weekly, monthly or yearly.

The sequence of events is not brought out on the DFD.

TYPES OF DATA FLOW DIAGRAMS

Current Physical

Current Logical

New Logical

New Physical

**CURRENT PHYSICAL:**

In Current Physical DFD proecess label include the name of people or their positions or the names of computer systems that might provide some of the overall system-processing label includes an identification of the technology used to process the data. Similarly data flows and data stores are often labels with the names of the actual physical media on which data are stored such as file folders, computer files, business forms or computer tapes.

**CURRENT LOGICAL:**

The physical aspects at the system are removed as mush as possible so that the current system is reduced to its essence to the data and the processors that transform them regardless of actual physical form.

**NEW LOGICAL:**

This is exactly like a current logical model if the user were completely happy with he user were completely happy with the functionality of the current system but had problems with how it was implemented typically through the new logical model will differ from current logical model while having additional functions, absolute function removal and inefficient flows recognized.

**NEW PHYSICAL**:

The new physical represents only the physical implementation of the new system.

**RULES GOVERNING THE DFD’S**

**PROCESS**

No process can have only outputs.

No process can have only inputs. If an object has only inputs than it must be a sink.

A process has a verb phrase label.

**DATA STORE**

Data cannot move directly from one data store to another data store, a process must move data.

Data cannot move directly from an outside source to a data store, a process, which receives, must move data from the source and place the data into data store

A data store has a noun phrase label.

**SOURCE OR SINK**

The origin and /or destination of data.

Data cannot move direly from a source to sink it must be moved by a process

A source and /or sink has a noun phrase land

**DATA FLOW**

A Data Flow has only one direction of flow between symbol. It may flow in both directions between a process and a data store to show a read before an update. The later is usually indicated however by two separate arrows since these happen at different type.

A join in DFD means that exactly the same data comes from any of two or more different processes data store or sink to a common location.

A data flow cannot go directly back to the same process it leads. There must be atleast one other process that handles the data flow produce some other data flow returns the original data into the beginning process.

A Data flow to a data store means update ( delete or change).

A data Flow from a data store means retrieve or use.

A data flow has a noun phrase label more than one data flow noun phrase can appear on a single arrow as long as all of the flows on the same arrow move together as one package.

Data flow diagrams represent the flow of data through a system. A DFD is composed of:

Data movement shown by tagged arrows.

Transformation or process of data shown by named bubbles.

Sources and destination of data represented by named rectangles.

Static storage or data at rest denoted by an open rectangle that is named.

The DFD is intended to represent information flow but it is not a flow chart and it is not intended to indicate decision-making, flow of control, loops and other procedural aspects of the system. DFD is a useful graphical tool and is applied at the earlier stages of requirements analysis. It may be further refined at preliminary design states and is used as mechanism for creating a top level structural design for software.

The DFD drawn first at a preliminary level is further expanded into greater details:

The context diagram is decomposed and represented with multiple bubbles.

Each of these bubbles may be decomposed further and documented as more detailed DFD’s

**diagram level0 for Vehicle monitoring system**

View vehicle details

Order for vehicle

Get Registration Form

View servicing details

ADMIN

View User details

New Centers

Add New Vehicle

View details

View Orders

View the details

Payment registration

USER

Distributor

**LEVEL -1 DIAGRAM:**

*Dfd level1 for user module*

Payment mode

Servicing db

View servicing details

Payment db

Order db

Vehicle db

Order for vehicle

View vehicle details

User

Dfd level1 for distributor

View order details

Conformation

Db

Payment db

Order db

Login db

User confirmation

User pay ment details

Login

**Did level 1 for admin module**

Centre db

Add vehicle details

View registration details

Vehicle db

Vehicle registration db

Login db

Add servicing centers

Login

**UML DIGRAM FOR E-BANKING SYSTEM**

**What is UML?**

􀂄 UML – Unified Modeling Language

􀂄 Standard language for specifying, visualizing,

constructing and documenting the artifacts of

software systems.

􀂄 Collection of best engineering practices that

have proven successful in modeling large

and complex systems.

**What is UML - Goals**

􀂄 Provide users with a ready-to-use, expressive visual modeling

language so they can develop and exchange meaningful models.

􀂄 Provide extensibility and specialization mechanisms to extend the

core concepts.

􀂄 Be independent of particular programming languages and

development processes.

􀂄 Provide a formal basis for understanding the modeling language.

􀂄 Encourage the growth of the OO tools market.

􀂄 Support higher-level development concepts such as collaborations,

frameworks, patterns and components.

􀂄 Integrate best practices.

**Why use UML?**

􀂄 Helps to reduce cost and time-to-market.

􀂄 Helps managing a complex project

architecture.

􀂄 Helps to convey ideas between

developers\designers\etc.

**Background**

􀂄 **1970** – Object-oriented modeling languages began to

appear.

􀂄 **1996** – Release of UML 0.9 by by Grady Booch, Jim

Rumbaugh of Rational Software Corporation, Ivar

Jacobson of Objectory company.

􀂄 **1996** – Release of UML 1.0 by Digital Equipment, HP, ILogix,

IntelliCorp, IBM, ICON, MCI, Microsoft, Oracle,

Rational, TI and Unisys.

􀂄 **1997** – Release of UML 1.1 by IBM, ObjecTime,

Platinum, Ptech, Taskon, Reich and Softeam

􀂄 **2001** – Work on UML 2.0 specifications.

**UML Diagrams – con**

􀂄 **Structural diagrams –** Used to describe the building blocks of the system –

features that do not change with time.These diagrams answer the question .

What's there

**Behavioral diagrams –** Used to show

how the system evolves over time

(responds to requests, events, etc.)

**Use Case Diagrams** Describes what a system does from the standpoint of an

external observer

Emphasis on *what* a system does rather then *how*.

􀂄 **Scenario** – an example of what happens when someone

interacts with the system.

􀂄 **Actor** – A user or another system that interacts with the

modeled system.

􀂄 A use case diagram describes the relationships between

*actors* and *scenarios*.

􀂄 Provides system requirements from the user’s point of

view.

**Use Case Diagrams – cont.**

UML defines 3 kinds of associations:

**Association –** defines a relationship between an *actor* and a *use case*.

􀂅 **Extend -** defines that instances of a *use case* may beaugmented with some additional behavior defined in

an extending *use case*.

**Uses -** defines that a *use case* uses a behavior defined in another *use case*.

**Usecase for vehiclemonitoring system**

****

**Interaction Diagrams**

􀂄 Used to model the behavior of several objects in

a use case.

􀂄 Demonstrates collaboration between the

different objects.

􀂄 **Sequence Diagram** displays the time sequence

of the objects participating in the interaction.

􀂄 **Collaboration Diagram** displays an interaction

organized around the objects and their links to

one another.

**Sequence diagram for administrator:**



􀂄 **Collaboration Diagram for administrator**



**Sequence diagram for user module**



􀂄 **Collaboration Diagram for user module**



**Sequence diagram for distributor module**



􀂄 **Collaboration Diagram for distributor**



**Class Diagrams**

􀂄 Displays objects structure, contents and

relationships.

􀂄 Class diagrams are static – display what

interacts but not what happens when

interaction occurs.

**Class Diagrams – cont.**

􀂄 Classes are represented by a rectangle divided to three

parts: class name, attributes and operations.

􀂄 Attributes are written as:

visibility name [multiplicity] : type-expression = initial-value

􀂄 Operations are written as:

visibility name (parameter-list) : return type-expression

􀂄 Visibility is written as:

+ public

# protected

- private

**Class Diagrams –**

**Relationships**

􀂄 Class Diagrams have 3 kinds of

relationships:

􀂅 **Association** – Two classes are

associated if one class has to know

about the other.

􀂅 **Aggregation** – An association in

which one class belongs to a collection

in the other.

􀂅 **Generalization** – An inheritance link

indicating one class is a base class ofthe other.

**Dependency –** A labeled dependency

between classes (such as friend

classes, instaciation)



**Component diagram for vehicle monitoring system**

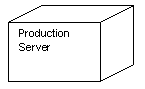
In component-based development (CBD), component diagrams offer architects a natural format to begin modeling a solution. Component diagrams allow an architect to verify that a system's required functionality is being implemented by components, thus ensuring that the eventual system will be acceptable.

In addition, component diagrams are useful communication tools for various groups. The diagrams can be presented to key project stakeholders and implementation staff. While component diagrams are generally geared towards a system's implementation staff, component diagrams can generally put stakeholders at ease because the diagram presents an early understanding of the overall system that is being built.



**Deployment diagram fro vehicle monitoring system**

**Node:** The element that provides the execution environment for the components of a system. Depicted by a cube with the name of the object in it, preceded by a colon, and underlined.



|  |  |
| --- | --- |
| **Connection:** Similar to the relation/association used in class diagrams to define the interconnection between nodes. | http://www.developer.com/img/articles/2003/12/22/UML11T02.gif |

The basic deployment diagram element is the node. The node represents the environment in which a component or a set of components execute. This means that a node in a deployment diagram can represent a multitude of things—physical hardware such as a server machine, a system software like an operating system, or even application infrastructure software like a Web server, application server, database server, and so forth. The different nodes in the deployment diagram can be interconnected to represent interdependencies, thus providing a deployment diagram that is easy to comprehend and provides the complete deployment environment of a system.



**Activity Diagram**

􀂄 Displays a workflow behavior of a system.

􀂄 Somewhat similar to a state diagram

􀂅Activities are states that represent the

performance of actions or subactivities.

􀂅Transitions are triggered by the completion of

actions or subactivities.

**Activity Diagram**

􀂄 Activity diagram notations:

􀂅 **Swimlane** – Used to organize responsibility for actions and

subactivities. Often corresponds to organizational units in a

business model.

􀂅 **Fork** - Splits an incoming transition into several concurrent

outgoing transitions. All of the transitions fire together.

􀂅 **Join** - Merges transitions from concurrent regions into a single

outgoing transition. All the transitions fire together.

􀂅 **Decision** – A state node that represents a decision. Each

transition from this node depends on a Boolean condition.

**Activity diagram for vehicle monitoring system:**

An activity diagram is a diagram that shows activities and actions to describe workflows. In the Unified Modeling Language an activity diagrams.

Flow chart on steroids. This diagram is great for expressing process flow. I like to create an activity diagram for each use case in the system. This helps me ensure that there are no loose ends in a use case's textual description.

**Activity for User module:**

View vehicle details

Order for vehicle

View servicing centre details

Get confirmation

Valid

N

Y

**Activity diagram for Distributor:**

Login

View order details

Send confirmation

Payment details

Send payment details

Valid

N

Y

**Activity diagram for administrator:**

Login

View payment details

Add service centre details

Add vehicle details

View vehicle details

Valid

N

Y

**E-R DIAGRAMS**

Entity Relationship Diagram give the structural representation of the relationship between entities in the system. This representation of relations gives the clear understanding how the data flow between entities. This diagram also explains how many members involved in the system.

The symbols we use in this representation is

Entity

Week Entity

Relationship

Attribute

**E-R diagram for administrator system**

Vehicle details

Manage

Manage

Manage

Admin

Manage

Vehicle info

Personal info

User

Service centre details

Registration info

Personal info

User

**E-R diagram for distributor**

Customer

Manage

Distributor

Manage

Order info

Personal info

Customer

Send confirmation

Payment details

**E- R DIAGRAM FOR USER MODULE**

Vehicle

Oder for

Services

Go for

Vehicle

Search for

user

get

Confirmation

**3.4 REQUIREMENTS**

**3.4.1 SOFTWARE REQUIREMENTS**

**Front End Tool**: Java Server Pages, Java Script, and HTML

**Back End Tool**: Oracle 9i

### 

### Software Interfaces

Operating System: Windows 2000

Database : Oracle 9i

Server : Tomcat 5.0

Explorer : Microsoft Internet Explorer

**3.4.2 HARDWARE REQUIREMENTS**

Processor : Pentium- III

Cache : 256 KB Cache

FDD : 1.44 MB Floppy drive

Hard Disk : 40 GB IDE Hard disk

RAM : 256 MB

Monitor : LG

**Hardware Interfaces**

Keyboard : 105 keys

Mouse : 3 button mouse

**3.3. DATA DICTIONARY**

The logical characteristics of current systems data stores, including name, description, aliases, contents, and organization, identifies processes where the data are used and where immediate access to information required, Serves as the basis for identifying database requirements during system design.

**Uses of Data Dictionary:**

1. To manage the details in large systems.
2. To communicate a common meaning for all system elements.
3. To Document the features of the system.
4. To facilitate analysis of the details in order to evaluate characteristics and determine where system changes should be made.
5. To locate errors and omissions in the system.

**TABLES**

**TABLE FOR SERVICE CENTER**

|  |  |  |
| --- | --- | --- |
| Brach id | Varchar | notnull |
| City name | Varchar | Notnull |
| Branch name | Varchar | Notnull |
| Branch addr | Varchar | Notnull |
| Create date | Datetime | notnull |

**TABLE FOR VEHICLE DETAILS**

|  |  |  |
| --- | --- | --- |
| VEHICLE id | Varchar | notnull |
| VEHICLE name | Varchar | Notnull |
| FUEL TYPE | Varchar | Notnull |
| ECONAMY | Varchar | Notnull |
| GEARBOX | Datetime | notnull |
| ENGINE TYPE | Datetime | Notull |
| Fuel reserve | Varchar | Notnull |
| Clutch | Varchar | Notnul |
| Vehiclemodel | Varchar | Notnull |
| Displace ment | Varchar | Notnull |
|  |  |  |
|  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| up id | Varchar | notnull | Primary key |
| User id | Varchar | Notnull | Foreign key |
| First name | Varchar | Notnull |  |
| Last name | Varchar | Notnull |  |
| applied date | Datetime | notnull |  |
| Email id | Varchar | Notnull |  |
| Cdate | Datetime | Notnull |  |

Login table

|  |  |  |  |
| --- | --- | --- | --- |
| User id | Varchar | Notnull | primary key |
| User name | Varchar | Notnull |  |
| Password | Varchar | Notnull |  |
| Create date | Datetime | notnull |  |

**6. TESTING**

#### What is Testing?

A process of executing a program with the explicit intention of finding errors, that is making the program fail.

**Software Testing:**

It is the process of testing the functionality and correctness of software by running it. Process of executing a program with the intent of finding an error.

A good test case is one that has a high probability of finding an as yet undiscovered error. A successful test is one that uncovers an as yet undiscovered error. Software Testing is usually performed for one of two reasons:

* Defect detection
* Reliability estimation

**Black Box Testing:**

Applies to software systems or module, tests functionality in terms of inputs and outputs at interfaces.Test reveals if the software function is fully operational with reference to requirements specification.

**White Box Testing:**

Knowing the internal workings i.e., to test if all internal operations are performed according to program structures and data structures.

To test if all internal components have been adequately exercised.

**Software Testing Strategies:**

A strategy for software testing will begin in the following order:

1. Unit testing
2. Integration testing
3. Validation testing
4. System testing

**Unit testing**

It concentrates on each unit of the software as implemented in source code and is a white box oriented. Using the component level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. In the unit testing,

The step can be conducted in parallel for multiple components.

**Integration testing:**

Here focus is on design and construction of the software architecture. Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

**Validation testing:**

In this, requirements established as part of software requirements analysis are validated against the software that has been constructed i.e., validation succeeds when software functions in a manner that can reasonably expected by the customer.

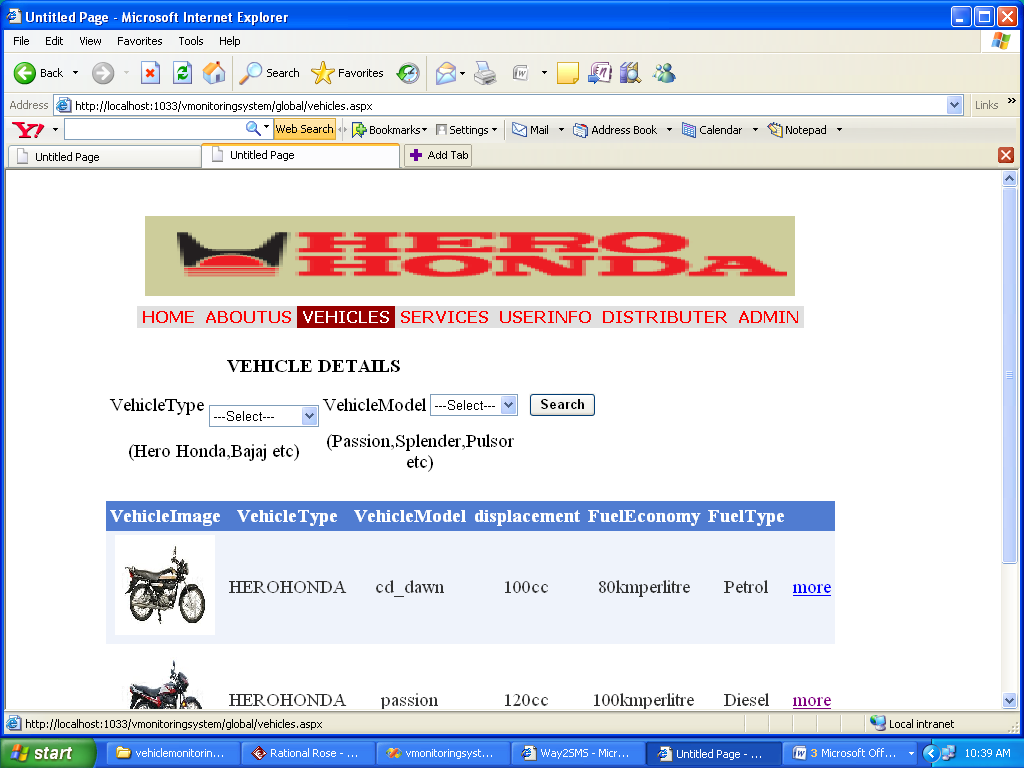
**System testing:** In this software and other system elements are tested as a whole.

Home page for vehicle monitoring system

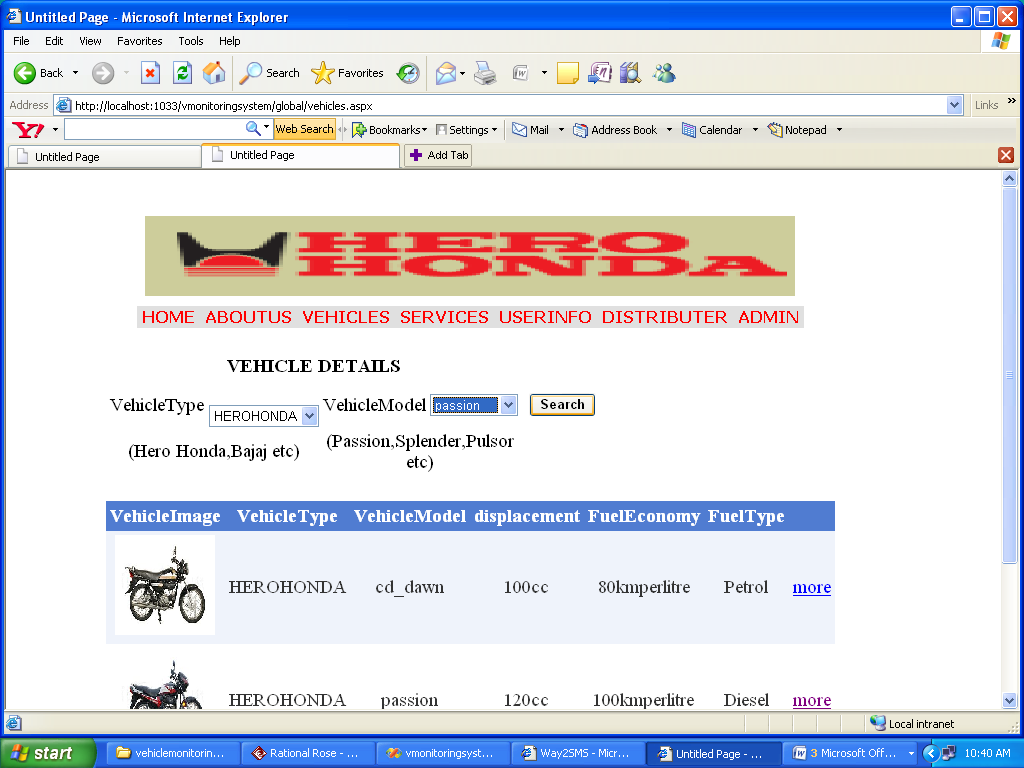
About us page

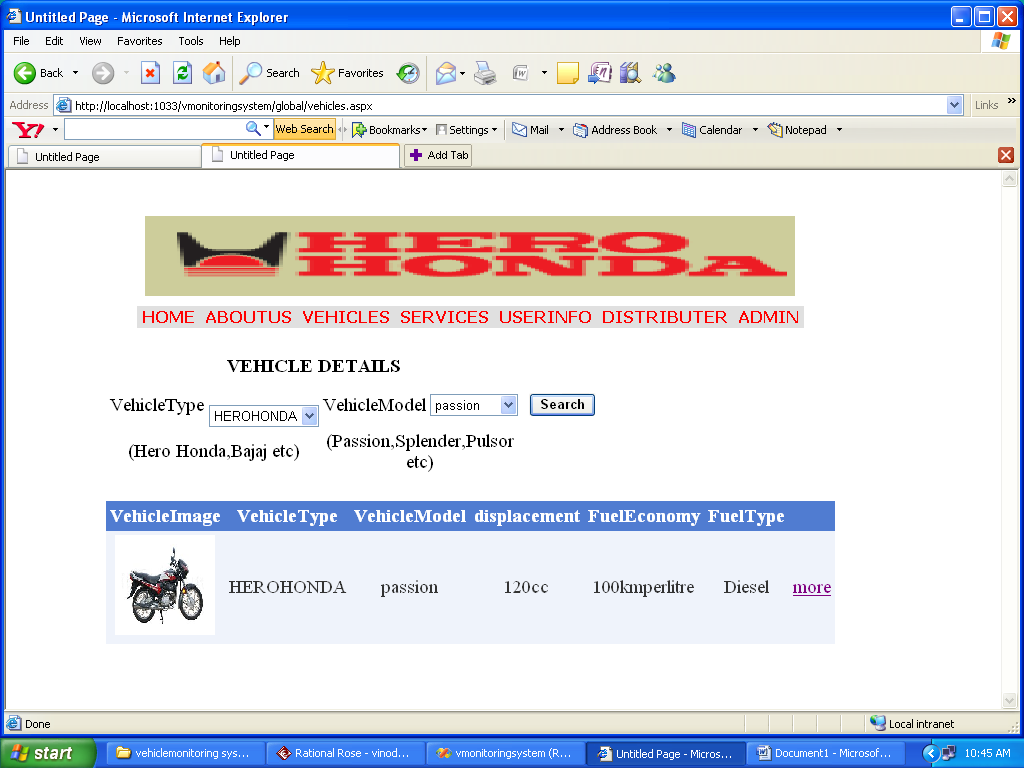


User can view the vehicle details here

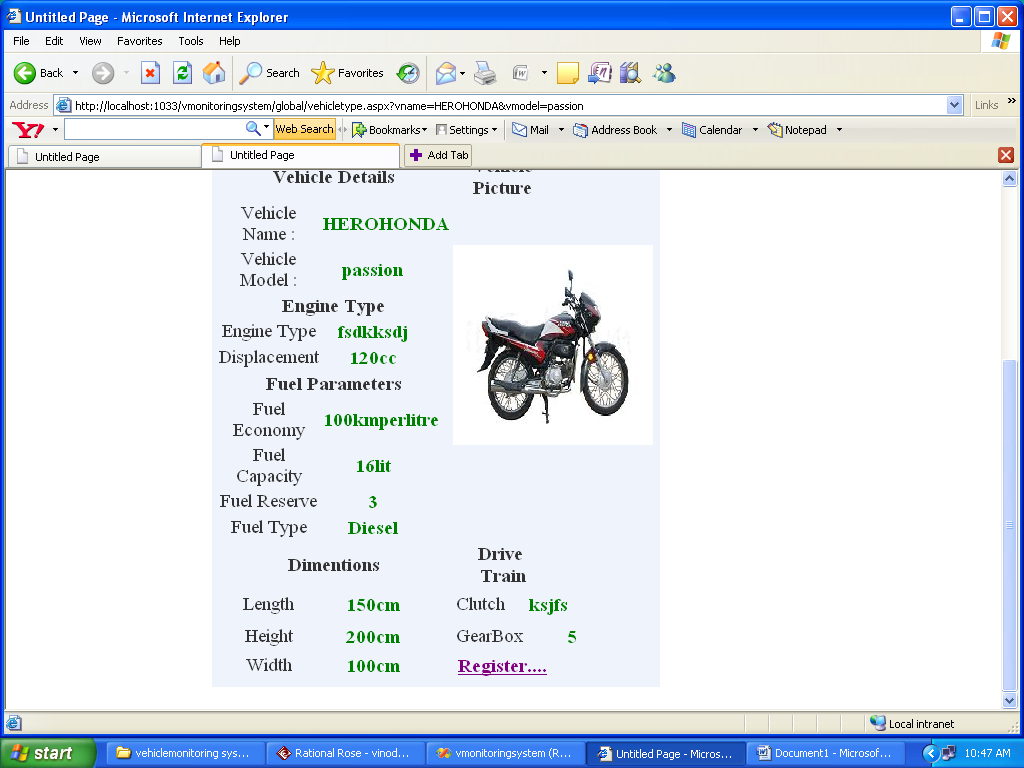


User can search for particular vehicle details

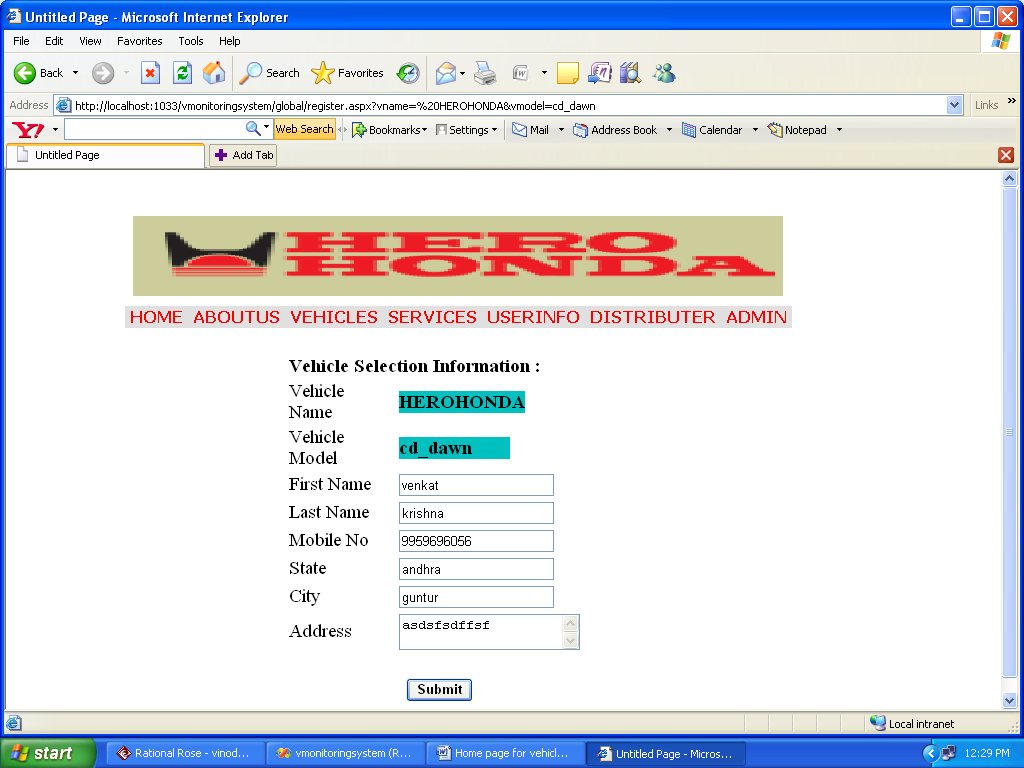


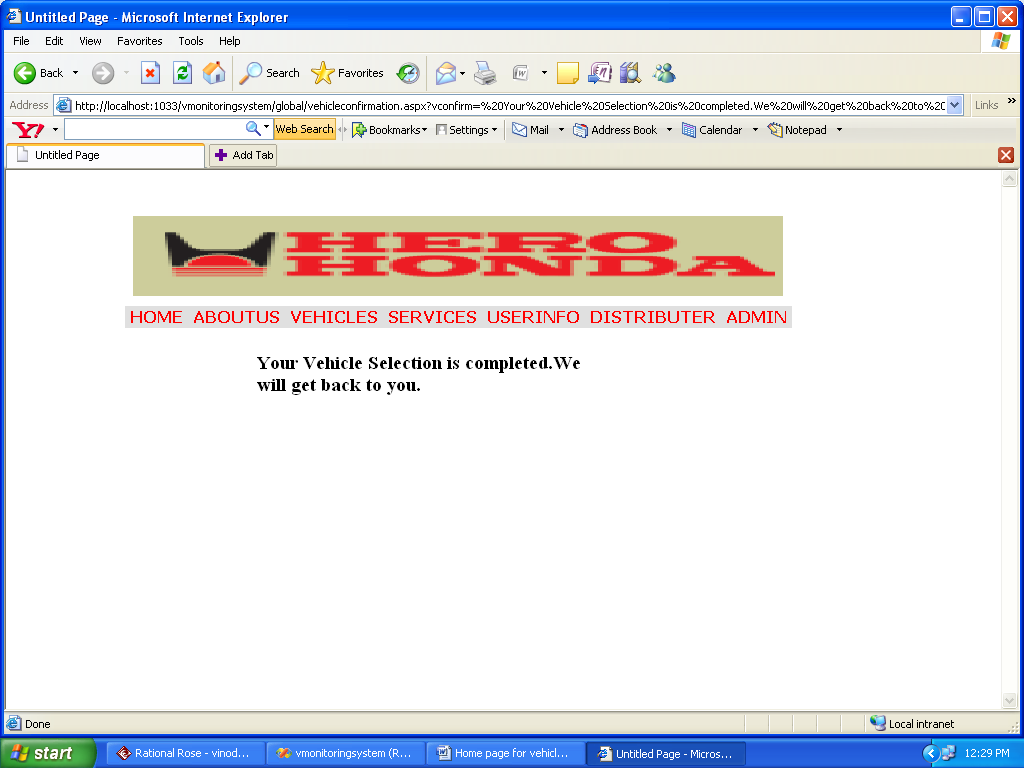


User wants to total details about his particular vehicle

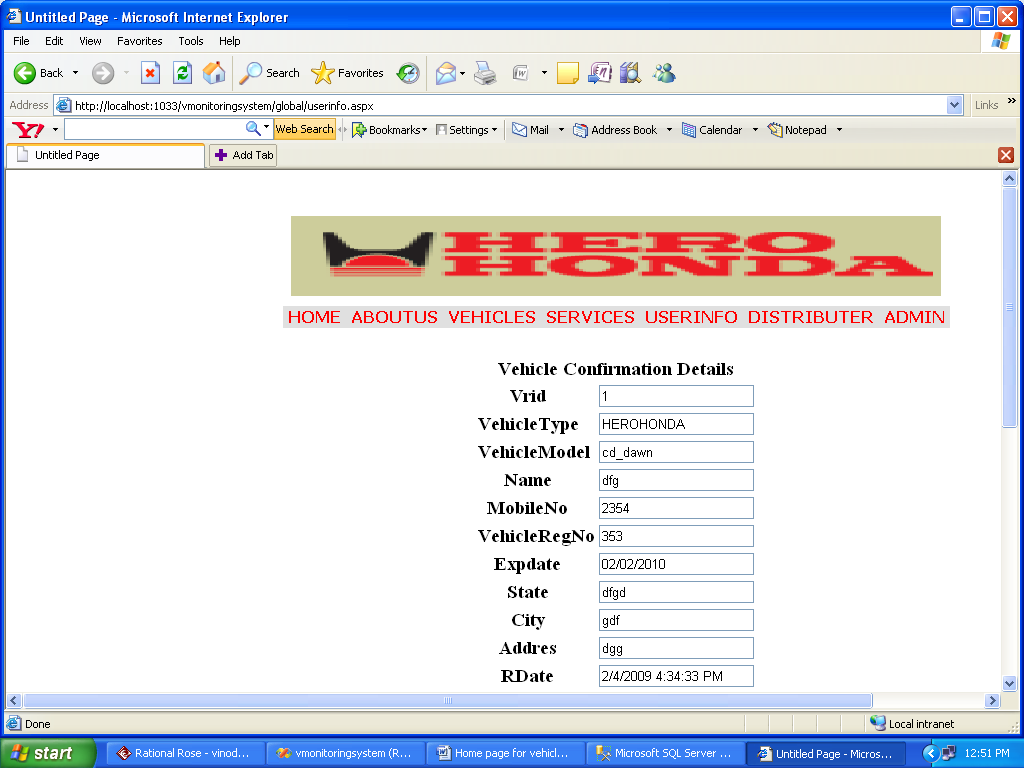


He can send a request for for his selection



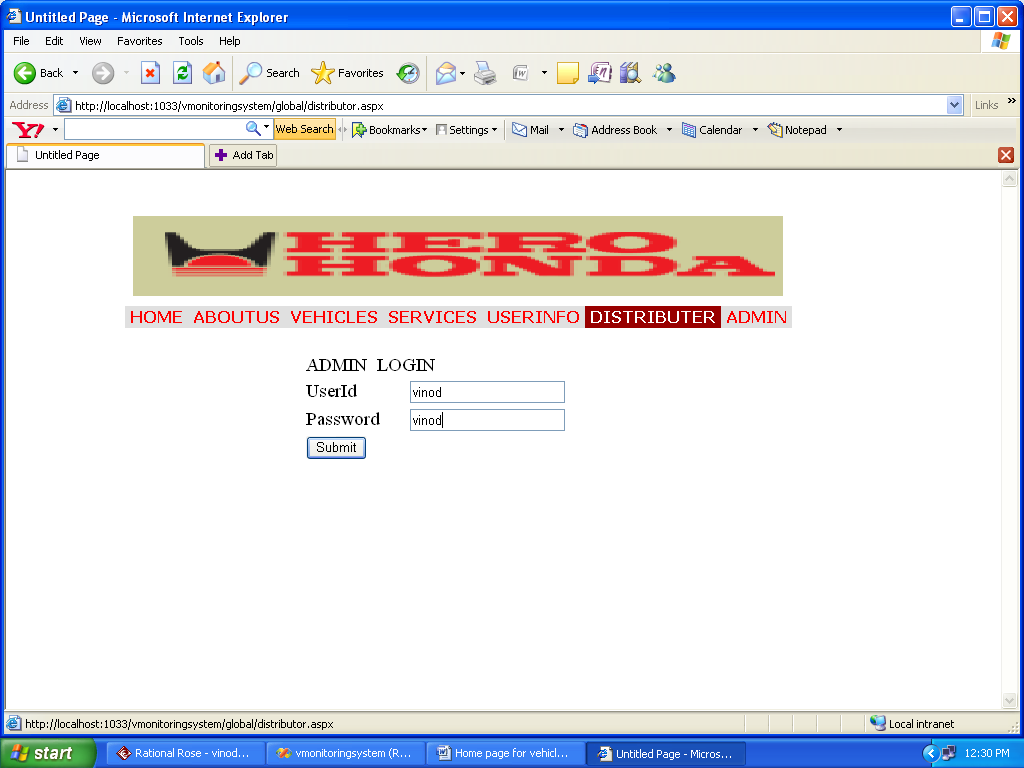


Users can his confirmation details in user info page

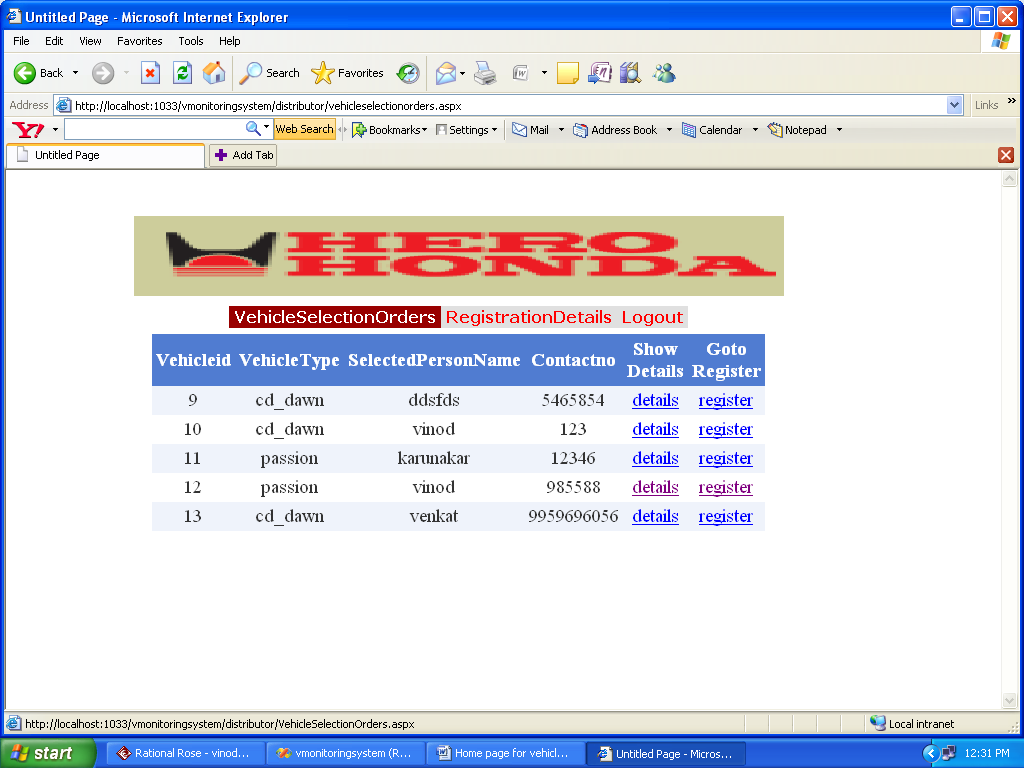


User can view servicing centre details



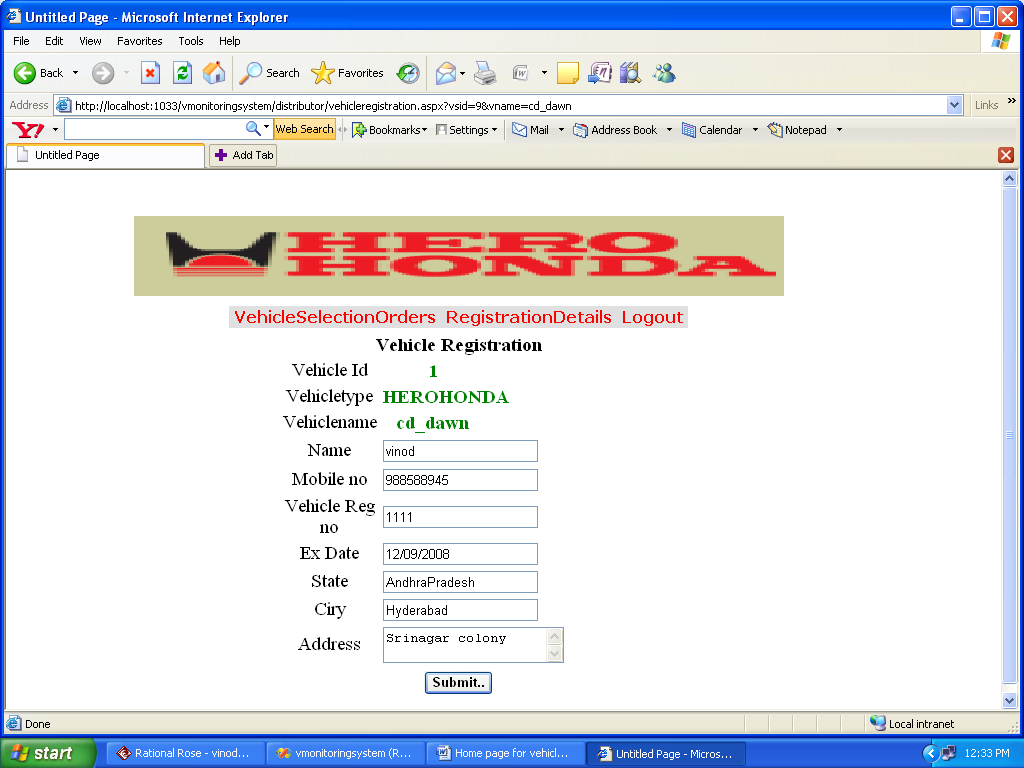


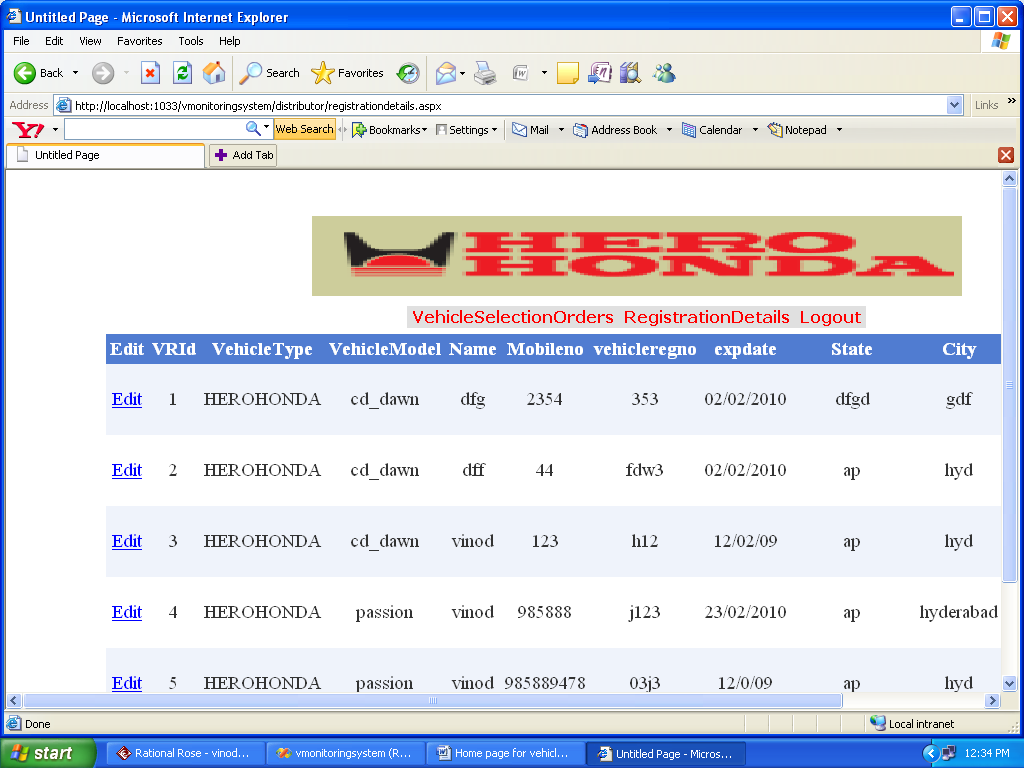
Distributor can view selection order details



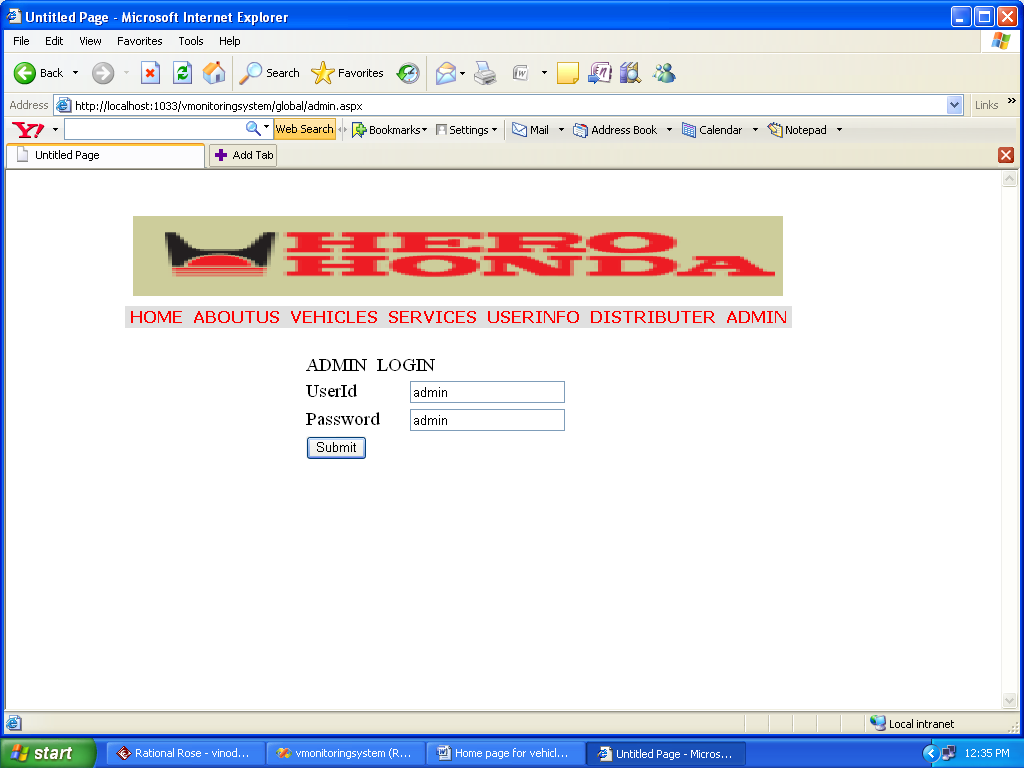


Distributor payment details





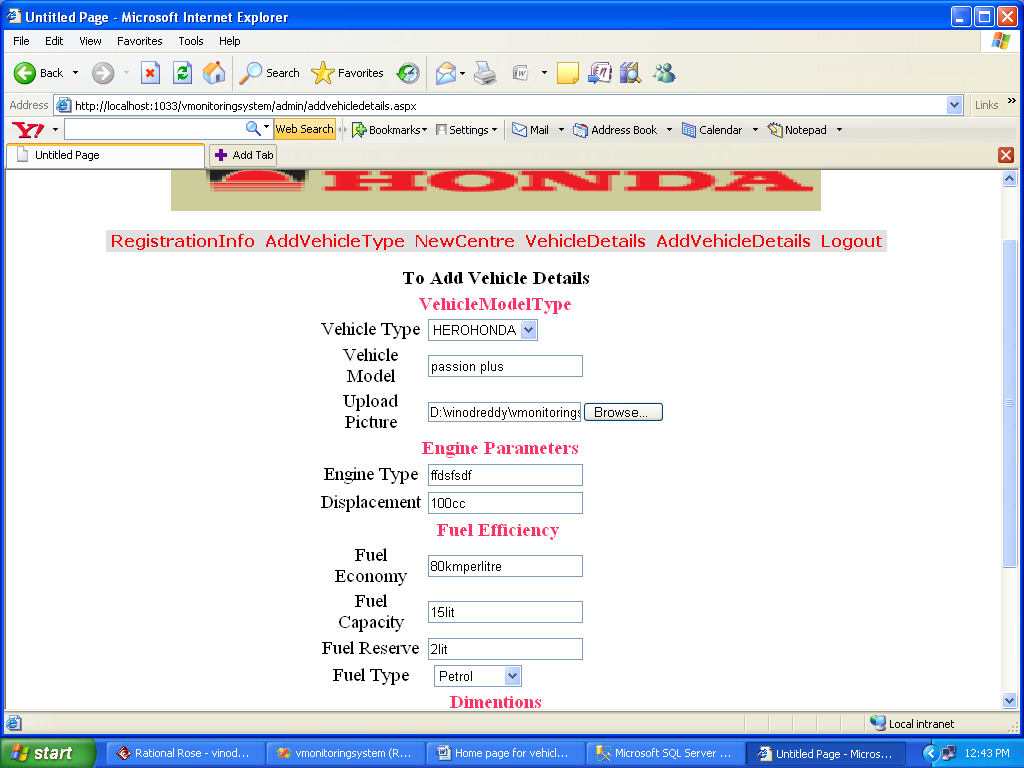
Admin login

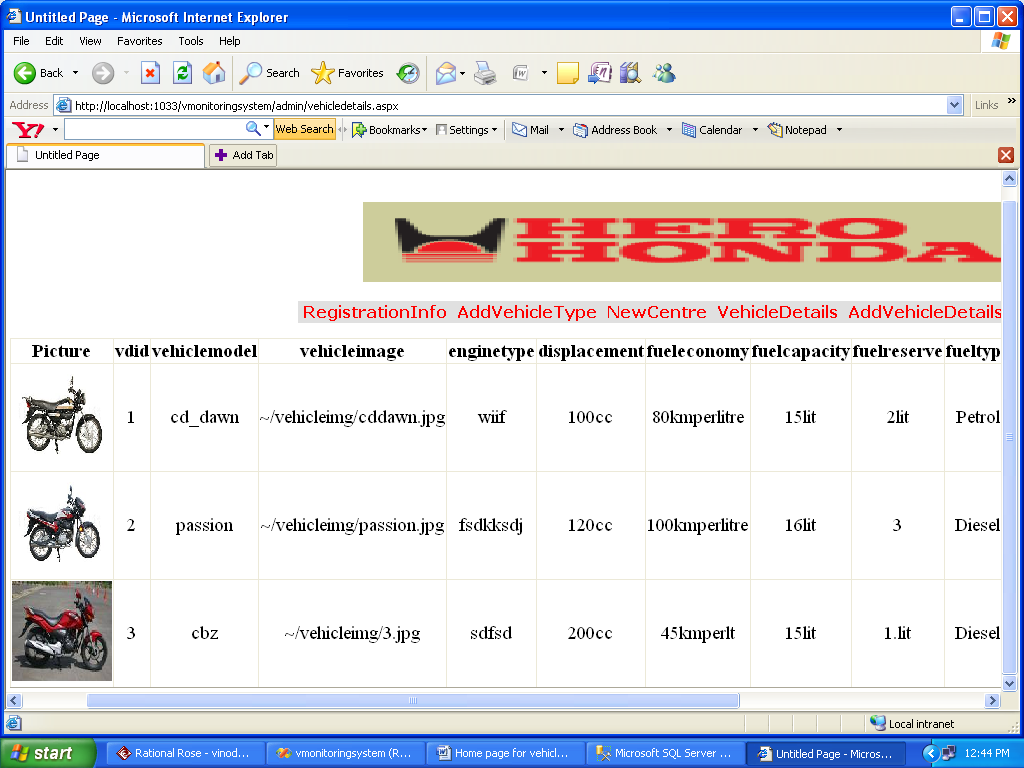


Admin view registered details



Admin adding vehicle details



Admin can view vehicles details

Admin add new centre details

