NEAR EAST UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

Graduation Project

COM – 400

SOFTWARE DESIGN FOR MONITORING THE PARAMETER OF ENVIRONMENTS : JAVA APPLICATION

STUDENT NO : 20010404

STUDENT NAME : AYKUT DANIŞMAN

SUPERVISOR: : ASSIST.PROF.DR. ADIL AMIRJANOV

LEFKOŞA – 2006

ACKNOWLEDGMENT

First of all, I would like to express my thanks to my supervisor Mr. Adil for supervising my project. Under the guidance of him, I successfully overcome many difficulties and I learned a lot from him. The great one is to solve the problems by my own. He never give me the direct solution, instead he always try to teach me how to create my own solution for a case. I will always be greatfull for this.

I also want to thanks to all my friends and specially to Muhammed Akgün and Caner Çakır who gave me a great supports.

Finally, my family. If they don't beliave on me, I won't be here. I will live for them.

ABSTRACT

The aim of this project is designing a software with the approach of object oriented programming by using Unified Modeling Language (UML) and constructing this design with JAVA Programming Language.

Simply the program is a simulation of pressure and the temperature measurement for an area from several different gauges. In the structure of the program, each gauges presented as an object as it should be in object oriented programming model. Number of gauges, which is decided by the operator, used in the experiment are created as objects dynamically and can not be more then 5. Each created gauges objects should get the temperature and pressure from its assigned sensor and do the necessary operations to draw the graphics of each gauges in each seconds. Also for each average period, that has been decided by the operator, the average is taken for that period and the graphics is drawn. This process is repeated for every time period until the experiment time over.

For all these operations to be done for each gauge separately and for each second, Thread technique is used.

INTRODUCTION

The Internet is growing by leaps and bounds. This incredible growth is the pathway to opportunity. Everybody who's anybody in the world of telecommunications is looking for ways to enhance the Internet's online experience. One company that has scored a big hit on the Internet is Sun Microsystems, who recently released an unusual programming language called Java.

What's so special about Java? Java enables programmers to create something called applets. Applets are special computer programs that can be included as an element of a Web page. When the user views a Web page containing one of these applets, the machine he's connected to automatically sends the applet to the user and the user's own Java-compatible browser runs the applet. Because applets are transferred in a non-machine-specific form, they can run on any machine that has a Java interpreter.

Java, can also be used in developing normal executable applications. Java Applications that have already been released include games, spreadsheets, graphing programs, animation controllers, simulators, and much, much more. Java is so intriguing and so successful that even major players in the industry, including Netscape and Microsoft, have jumped aboard, providing Java-compatible software for the Internet.

This project is made by Java Application and used several techniques which are extreme Java futures. Chapter-1 is about overall Java concepts like; how to develop a Java Application, programming the user interface, handling events and exception handling. It is tried to give some overall information about the concepts that are used in the project. In chapter-2, it is mentioned about the great future of Java, Threads. What is Thread? How they are work? And how they can be constructed. In chapter-3, levels of developing a software with the approach of object oriented programming is discussed with the visual notation technique UML. Finally in chapter-4 the Monitoring Software development process discussed deeply.

CHAPTER-1
JAVA FUNDAMENTALS

1.1 Introduction

1.1.1 Java and C++

Java is largely based on the C++ programming language. C++ is one of the most popular programming languages for a variety of platforms and is a high-level, object-oriented programming language. Unfortunately, C++ is a very difficult language. It is designed for high-level commercial applications; as a result, it is very powerful and gives the programmer access to a very detailed level of machine control to optimize performance. The level of control that C++ provides is often unnecessary for midrange applications and can be very difficult for an inexperienced programmer to manage.

Java is designed to include the functions of a high-level programming language while eliminating some of the more difficult aspects of coding in C++. Keep in mind that Java is a new language that is based on some of the more popular programming languages of today such as C and C++. Java takes its functionality from these languages, but tries to improve on their implementation whenever possible.

1.1.2 A Platform-Independent Solution

Java has been molded to fit many different projects since it was first created in the early 1990s. It has gone from a programming language for personal digital assistants to a programming language or interactive TV set-top boxes, and then to its final incarnation as a Web programming language. This transformation should give you some indication of Java's flexibility and portability; it is designed to be machine-independent and function within different operating systems. Portability is one of Java's principal goals. Java code is designed to be platform-independent and has several features designed to help it achieve that goal.

When you write a Java application, you are writing a program that is designed to be run on a very special computer: the Java Virtual Machine. The Java Virtual Machine is the first step towards a platform-independent solution. When you are developing software in a language like C++, you generally program for a specific platform, such as a Windows machine or a Macintosh. The programming language will use a variety of functions that are very specific to whatever processor is used by the machine you are programming for. Because the language uses machine-specific instructions, you have to modify your program for a new processor if you want to run your program on another machine. But, Java code is not written for any type of physical computer. Instead, it is written for a special computer called the Virtual Machine, which is really another piece of software. The Virtual Machine then interprets and runs the Java program you have written. The Virtual Machine is programmed for specific machines, so there is a Windows 95 Virtual Machine, a Sun Virtual Machine, and so on. There is even a copy of the Virtual Machine built into Netscape, allowing the browser to run Java programs.

1.1.3 One Binary Fits All

The Virtual Machine requires special binary code to run Java programs. This code, called *byte code*, does not contain any platform-specific instructions. That means if you write a program on a Sun workstation and compile it, the compiler will generate the same bytecode as a machine running Windows 95. This code is the second step towards a platform-independent development environment. Imagine the time and money that could be saved if software today could be written once and used on all computers. There would be no need for Mac or Windows versions of software and no need to hire hordes of programmers to convert some Mac program to a Windows version.

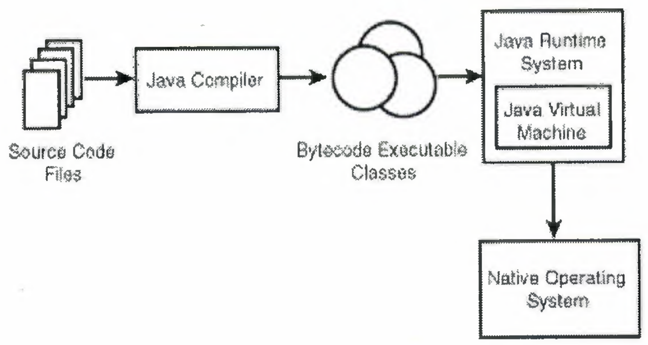


Figure 1.1 Compiling Process

1.2 Object-Oriented Programming

In classic, procedural programming you try to make the real world problem you're attempting to solve fit a few, pre-determined data types: integers, floats, Strings, and arrays perhaps. In object oriented programming you create a model for a real world system. Classes are programmer-defined types that model the parts of the system.

A class is a programmer defined type that serves as a blueprint for instances of the class. You can still have ints, floats, Strings, and arrays; but you can also have cars, motorcycles, people, buildings, clouds, dogs, angles, students, courses, bank accounts, and any other type that's important to your problem.

Classes specify the data and behavior possessed both by themselves and by the objects built from them. A class has two parts: the fields and the methods. Fields describe what the class is. Methods describe what the class does.

Using the blueprint provided by a class, you can create any number of objects, each of which is called an instance of the class. Different objects of the same class have the same fields and methods, but the values of the fields will in general differ. For example, all humans have eye color but the color of each human's eyes can be different from others.

On the other hand, objects have the same methods as all other objects in the class except in so far as the methods depend on the value of the fields and arguments to the method.

This dichotomy is reflected in the runtime form of objects. Every object has a separate block of memory to store its fields, but the bytes in the actual methods are shared between all objects in a class.

Another common analogy is that a class is to an object as a cookie cutter is to a cookie. One cookie cutter can make many cookies. There may be only one class, but there can be many objects in that class. Each object is an instance of one class.

1.2.1 Flexible Object-Oriented Design

Most of the thought involved in creating any software goes toward solving problems during design. With a good design, the actual coding becomes largely mechanical.

In general, design patterns are not code. Instead, they collect a common interaction and a common set of objects at an abstract level. The actual coding depends on the particular problem. Design patterns are not a template mechanism, like the templates in C++. They are also not classes that you can directly apply. Moreover, design patterns are not language-specific. The pattern itself is completely abstract. It is the kernel of a solution, not the solution itself. In the case of Java, there are classes and interfaces that allow you to apply some patterns easily, but this is an instance of a pattern. This distinction-between the pattern and one implementation of the pattern-is the same as the distinction between a class and an instance of that class. One is abstract, describing how to construct something. The second is the product of that construction.

1.2.2 Encapsulation

One major difference between conventional structured programming and object-oriented programming is a handy thing called encapsulation. Encapsulation enables you to hide, inside the object, both the data fields and the methods that act on that data. (In fact, data fields and methods are the two main elements of an object in the Java programming language.) After you do this, you can control access to the data, forcing programs to retrieve or modify data only through the object's interface. In strict object-oriented design, an object's data is always private to the object. Other parts of a program should never have direct access to that data.

How does this data-hiding differ from a structured-programming approach? After all, you can always hide data inside functions, just by making that data local to the function. A problem arises, however, when you want to make the data of one function available to other functions. The way to do this in a structured program is to make the data global to the program, which gives any function access to it. It seems that you could use another level of scope-one that would make your data global to the functions that need it-but still prevent other functions from gaining access. Encapsulation does just that. In an object, the encapsulated data members are global to the object's methods, yet they are local to the object. They are not global variables.

1.2.3 Classes as Data Types

Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects. An object is just an instance of a data type. For example, when you declare a variable of type int, you're creating an instance of the int data type. A class is like a data type in that it is the blueprint upon which an object is based. When you need a new object in a program, you create a class, which is a kind of template for the object. Then, in your program, you create an instance of the class. This instance is called an object.

Classes are really nothing more than user-defined data types. As with any data type, you can have as many instances of the class as you want. For example, you can have more than one window in a Windows application, each with its own contents.

For example, think again about the integer data type (int). It's absurd to think that a program can have only one integer. You can declare many integers, just about all you want. The same is true of classes. After you define a new class, you can create many instances of the class. Each instance (called an object) normally has full access to the class's methods and gets its own copy of the data members.

1.2.4 Inheritance

Inheritance enables you to create a class that is similar to a previously defined class, but one that still has some of its own properties. Consider a car-simulation program. Suppose that you have a class for a regular car, but now you want to create a car that has a high-speed passing gear. In a traditional program, you might have to modify the existing code extensively and might introduce bugs into code that worked fine before your changes. To avoid these hassles, you use the object-oriented approach: Create a new class by inheritance. This new class inherits all the data and methods from the tested base class. You can control the level of inheritance with the public, private, and protected keywords.

1.2.5 Creating a Subclass

You derive a new class from a superclass by using the 'extends' keyword, an apt name for a keyword because by deriving a new class you usually extend the abilities of the original class. For example, when you create a new Frame, you start the Frame's class with a line that looks something like this:

```
public class MyApplet extends Applet
```

First, Java insists that all Frame classes be declared as public. Next in the line, you can see the class keyword followed by the name of the class. Finally comes the extends keyword followed by the name of the superclass. In English, the above line means that the public class MyApplet is derived from (is a subclass of) the existing Frame class. As you've probably already figured out, Frame is a class that the Java makers created for you. This class contains all the basic functionality you need to create an application. You need only extend the specifics of the class in order to create your own frame class.

1.2.6 Polymorphism

The last major feature of object-oriented programming is polymorphism. By using polymorphism, you can create new objects that perform the same functions as the base object but which perform one or more of these functions in a different way. For example, you may have a shape object that draws a circle on the screen. By using polymorphism, you can create a shape object that draws a rectangle instead. You do this by creating a new version of the method that draws the shape on the screen. Both the old circle-drawing and the new rectangle-drawing method have the same name (such as DrawShape()) but accomplish the drawing in a different way.

1.3 Applications & Applets

In essence, applets and Java applications share many of the same resources and features. The major anatomical difference between an applet and an application that makes both of them unique is that an applet extends the java.applet.Applet class, and an application utilizes the main() method. On a higher level, an application is not as restricted as an applet in terms of what it can access. On the other hand, applications cannot be embedded into HTML pages for any Internet user to view with a Java-capable browser. Applets and applications overlap in many areas, but they are specifically designed for different environments.

1.3.1 Java Applications

Java applications are different from Java applets in several ways. On the inside (that is, the source code level), you do not need to extend any class in the Java class library when creating an application. As a result, you do not have a set of basic methods that you override to execute your program. Instead, you create a set of classes that contain various parts of your program, and then you attach a main() method that will contain the code used to execute your code. The main() method is very similar to that of the C/C++ model; it is here that Java will start the actual execution of your application. The following shows what the structure for a typical Java application would look like this:

```
public class TheClass {
   /* variables and methods specific to the class TheClass
   are located here. */
   class ABClass {
      /* The body of the class ABClass
      is located here */
   }

   public static void main(String args[]) {
   /* The actual execution of the application
   is located here. **/
   }
}
```

The main() method shown in the preceding code is the system method that is used to invoke the application. As mentioned earlier, any action code should be located in main(). The main() method is more than just another method in a Java application. If you do not include a main() method in your application when you attempt to run it, you will receive an error message.

1.3.2 Windowed Applications

For an application to be framed, first your class must extends to Frame class. So when the constructor of this class called, means when an instance of that class is created, also a new frame is created. As it is mention before, an application must have a public class. This public class can be assumed as the main class of our application and must have a special method 'main'. When the application starts, the java run time environment looks for this main method in the public class and starts the execution from here. That is why we call the constructor of the class in main method. The rest of the instructions are done in the constructor and other user defined methods.

1.4 Programming User Interface-Java.awt

The java.awt package contains what is known as the Java Abstract Windowing Toolkit. The classes within this package make up the pre-built graphical user interface components that are available to Java developers through the Java Developer's Kit. Classes defined within this package include such useful components as colors, fonts, and widgets, such as buttons and scrollbars. This package also defines two interfaces, LayoutManager and MenuContainer, as well as the exception AWTException and the error AWTError. The java.awt package also contains two sub packages: java.awt.image and java.awt.peer.

Here is the hierarchy of the classes commonly used in building interfaces;

- AWT classes that inherit from Java.lang object.
- All the classes that control the placement of objects on the screen inherit from the Java.lang object.class.
- The Applet.class inherits from java.awt.Panel,so you can draw directly to an applet.

These class structures might seem a bit intimidating at first, but if you glance through them, you'll see that most AWT classes just inherit from Object. In addition, all the interactive elements (except menus) inherit from Component. The only other very important thing to note is that because Frame inherits from Window (which inherits from Container), Frames can directly contain other objects such as buttons, canvases, and so on.

1.4.1 The update, paint, and repaint Methods

You'll encounter three key methods over and over again as you work with the various user interface elements.

Repaint	→ Requests a redraw of an item or an entire interface. It then calls update.
Update	→ Controls what happens when repaint is called; you can override this method.
Paint	→ Determines what is done when any item is redrawn. It's called whenever something needs to be redrawn-for example, when a window is uncovered. All displayable entities have paint methods that they inherit from Component.

All these methods are defined in the Component class. This means that all the various interactive controls and Frames inherit these methods. Although all these methods can be overridden, you'll find that paint and update are the ones you work with most often. Interactive elements, such as buttons, are redrawn automatically; you don't have to implement a paint method for them.

1.4.2 Graphics

The Graphics class is an abstract class that provides the means to access different graphics devices. It is the class that lets you draw on the screen, display images, and so forth. Graphics is an abstract class because working with graphics requires detailed knowledge of the platform on which the program runs. The actual work is done by concrete classes that are closely tied to a particular platform. Your Java Virtual Machine vendor provides the necessary concrete classes for your environment. You never need to worry about the platform-specific classes; once you have a Graphics object, you can call all the methods of the Graphics class, confident that the platform-specific classes will work correctly wherever your program runs.

You rarely need to create a Graphics object yourself; its constructor is protected and is only called by the subclasses that extend Graphics. How then do you get a Graphics object to work with? The sole parameter of the Component.paint() and Component.update() methods is the current graphics context. Therefore, a Graphics object is always available when you override a component's paint() and update() methods. You can ask for the graphics context of a Component by calling Component.getGraphics(). However, many components do not have a drawable graphics context. Canvas and Container objects return a valid Graphics object; whether or not any other component has a drawable graphics context depends on the run-time environment. This restriction isn't as harsh as it sounds. For most components, a drawable graphics context doesn't make much sense; for example, why would you want to draw on a List? If you want to draw on a component, you probably can't. The notable exception is Button, and that may be fixed in future versions of AWT.

1.4.3 Images

An Image is a displayable object maintained in memory. To get an image on the screen, you must draw it onto a graphics context, using the drawImage() method of the Graphics class. For example, within a paint() method, you would call g.drawImage(image, ... , this) to display some image on the screen. In other situations, you might use the createImage() method to generate an offscreen Graphics object, then use drawImage() to draw an image onto this object, for display later.

The last argument 'this', of drawImage() is always an image observer--that is, an object that implements the ImageObserver interface. Call to drawImage() starts a new thread that loads the requested image. An image observer monitors the process of loading an image; the thread that is loading the image notifies the image observer whenever new data has arrived.

The Component class implements the ImageObserver interface; when you're writing a paint() method, you're almost certainly overriding some component's paint() method; therefore, it's safe to use this as the image observer in a call to drawImage(). More simply, we could say that any component can serve as an image observer for images that are drawn on it.

1.4.4 Drawing

The methods and tools described below enable you to draw simple graphical items that are non-interactive. All these operations are implemented as methods on the Graphics class. Because anything you can draw on has a Graphics class as an attribute, this doesn't limit the utility of these functions.

1.4.4.1 Graphics Methods

The class hierarchy for the Graphics class derives from the class java.lang.Object. The Graphics class represents the base class for all types of graphics contexts. The classes that are used in the project.

- **protected Graphics ()**
  Because Graphics is an abstract class, it doesn't have a visible constructor. The way to get a Graphics object is to ask for one by calling getGraphics() or to use the one given to you by the Component.paint() or Component.update() method.

  In addition to using the graphics contexts given to you by getGraphics() or in Component.paint(), you can get a Graphics object by creating a copy of another Graphics object. Creating new graphics contexts has resource implications. Certain platforms have a limited number of graphics contexts that can be active. For instance, on Windows 95 you cannot have more than four in use at one time.

Therefore, it's a good idea to call dispose() as soon as you are done with a Graphics object. Do not rely on the garbage collector to clean up for you.

- **public abstract FontMetrics getFontMetrics (Font font)**
  This version of getFontMetrics() returns the FontMetrics for the Font font instead of the current font. You might use this method to see how much space a new font requires to draw text.
    - **public int getAscent ()**
      The getAscent()method retrieves the space above the baseline required for the tallest character in the font. The units for this measurement are pixels. You cannot get the ascent value for a specific character.
    - **public int stringWidth (String string)**
      The stringWidth() method calculates the advance width of the entire string In pixels. Among other things, you can use the results to underline or center text within an area of the screen.
- **public abstract void setFont (Font font)**
  The setFont() method changes the current Font to font. If font is not available on the current platform, the system chooses a default. To change the current font to 12 point bold TimesRoman:
  ```
          setFont (new Font ("TimesRoman", Font.BOLD, 12));
  ```
- **public abstract void setColor (Color color)**
  The setColor() method changes the current drawing color to color. Color class defines some common colors for you. If you can't use one of the predefined colors, you can create a color from its RGB values.
  ```
  setColor (Color.red);
  ```
- **public abstract void drawLine (int x1, int y1, int x2, int y2)**
  The drawLine() method draws a line on the graphics context in the current color from (x1, y1) to (x2, y2). If (x1, y1) and (x2, y2) are the same point, you will draw a point. There is no method specific to drawing a point.
- **public void drawRect (int x, int y, int width, int height)**

The drawRect() method draws a rectangle on the drawing area in the current color from (x, y) to (x+width, y+height). If width or height is negative, nothing is drawn.

- **public abstract void clearRect (int x, int y, int width, int height)**
  The clearRect() method sets the rectangular drawing area from (x, y) to (x+width-1, y+height-1) to the current background color. The second pair of parameters is not the opposite corner of the rectangle, but the width and height of the area to clear.
- **public abstract void drawPolyline (int xPoints[], int yPoints[], int numPoints)**
  The drawPolyline() method functions like the 1.0 version of drawPolygon(). It plays connect the dots with the points in the xPoints and yPoints arrays and does not connect the endpoints. If either xPoints or yPoints does not have numPoints elements, drawPolygon() throws the run-time exception, ArrayIndexOutOfBoundsException.
- **public abstract void drawString (String text, int x, int y)**
  The drawString() method draws text on the screen in the current font and color, starting at position (x, y). The starting coordinates specify the left end of the String's baseline.
- **public abstract boolean drawImage (Image image, int x, int y, ImageObserver observer)**
  The drawImage() method draws image onto the screen with its upper left corner at (x, y), using observer as its ImageObserver. Returns true if the object is fully drawn, false otherwise.

1.5 Layouts

Layouts allow you to format components on the screen in a platform-independent way. Without layouts, you would be forced to place components at explicit locations on the screen, creating obvious problems for programs that need to run on multiple platforms. There's no guarantee that a TextArea or a Scrollbar or any other component will be the same size on each platform; in fact, you can bet they won't be. In an effort to make your

Java creations portable across multiple platforms, Sun created a LayoutManager interface that defines methods to reformat the screen based on the current layout and component sizes. Layout managers try to give programs a consistent and reasonable appearance, regardless of the platform, the screen size, or actions the user might take.

The standard JDK provides five classes that implement the LayoutManager interface. They are FlowLayout, GridLayout, BorderLayout, CardLayout, and GridBagLayout.

1.5.1 No Layout Manager

Although most containers come with a preset FlowLayout Manager, you can tell the AWT to use absolute positions with the following line of code:

```
setLayout(null);
```

This eliminates the default Layout Manager, or any other one that the container had been using, enabling you to position components by using absolute coordinates.

1.5.2 The GridLayout Manager

The GridLayout manager is used to lay out components in a grid of evenly spaced cells. The default constructor creates a grid with one column per component, but you can use other constructors to specify the exact number of rows and columns you need as well as the spacing between the cells. The GridLayout manager populates its cells from left to right and top to bottom.

1.6 AWT Components

1.6.1 Containers

A Container is a type of component that provides a rectangular area within which other components can be organized by a LayoutManager. Because Container is a subclass of

Component, a Container can go inside another Container, which can go inside another Container, and so on. Subclassing Container allows you to encapsulate code for the components within it. This allows you to create reusable higher-level objects easily. The containers that are most commonly used : JPanel, JFrame, JScrollPane, JOptionPane and JTabbedPane.

1.6.1.1 Frames

A Frame implements a resizable window that supports a menu bar, cursor, icon, and title. Frames generate the same events as windows, which they extend: WINDOW_DESTROY, WINDOW_ICONIFY, WINDOW_DEICONIFY, and WINDOW_MOVED.

The only parameter you can pass to the Frame constructor is a String, which will be the window title. It is generally created a class that extends Frame and contains event-handling methods that override Component methods such as ActionListener and ItemListener. When you extend the class, you can make creator methods with more input parameters.

Some Frame methods used in the project are;

- **public Frame ()**
  The constructor for Frame creates a hidden window with a window title of "Untitled" or an empty string. The default LayoutManager of a Frame is BorderLayout. DEFAULT_CURSOR is the initial cursor. To position the Frame on the screen, call Component.move(). Since the Frame is initially hidden, you need to call the show() method before the user sees the Frame.
- **public void setTitle (String title)**
  The setTitle() method changes the Frame's title to title.
- **public static void setDefaultLookAndFeelDecorated (Boolean Decoated)**
  If defaultLookAndFeelDecorated is true, the current LookAndFeel supports providing window decorations, and the current window manager supports undecorated windows, then newly created Frames will have their Window

decorations provided by the current LookAndFeel. Otherwise, newly created JFrames will have their Window decorations provided by the current window manager.

- **public void setSize(int width, int height)**
  Resizes this component so that it has width *width* and height *height*.
- **public void setVisible(boolean b)**
  Shows or hides this component depending on the value of parameter b.
- **public void dispose()**
  Releases all of the native screen resources used by this Window, its subcomponents, and all of its owned children.

1.6.1.2 Panels

Panel inherits from Container. It doesn't create its own window because it's used to group components inside other containers. Panels enable you to group items in a display in a way that might not be allowed by the available Layout Managers. If you have a number of entries in your interface, for example, that have a label and a text field, you can define a panel that contains a label and a text field and add the panel so that the label and the text field always stay together on the same line (which wouldn't be the case if you added the two items separately). Without the panel, the Layout Manager could put the label and the text field on different lines.

1.6.1.3 ScrollPane

A ScrollPane is a Container with built-in scrollbars that can be used to scroll its contents. In the current implementation, a ScrollPane can hold only one Component and has no layout manager. The component within a ScrollPane is always given its preferred size. While the scrollpane's inability to hold multiple components sounds like a deficiency, it isn't; there's no reason you can't put a Panel inside a ScrollPane, put as many components as you like inside the Panel, and give the Panel any layout manager you wish. The methods of ScrollPane used in the project are;

- **public JScrollPane(Component view, int vsbPolicy, int hsbPolicy)**
  Creates a JScrollPane that displays the view component in a viewport whose view position can be controlled with a pair of scrollbars. The scrollbar policies specify when the scrollbars are displayed, For example, if vsbPolicy is VERTICAL_SCROLLBAR_AS_NEEDED then the vertical scrollbar only appears if the view doesn't fit vertically. The available policy settings are listed at setVerticalScrollBarPolicy(int) and setHorizontalScrollBarPolicy(int).

1.6.1.4 JTabbedPane

A component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon. Tabs/components are added to a TabbedPane object by using the addTab and insertTab methods. A tab is represented by an index corresponding to the position it was added in, where the first tab has an index equal to 0 and the last tab has an index equal to the tab count minus 1.

The TabbedPane uses a SingleSelectionModel to represent the set of tab indices and the currently selected index. If the tab count is greater than 0, then there will always be a selected index, which by default will be initialized to the first tab. If the tab count is 0, then the selected index will be -1.

The methods of JTabbedPane used in the project are;

- **public void addTab(String title, Component component)**
  Adds a component represented by a title and no icon. Cover method for insertTab.
- **public void setEnabledAt(int index, boolean enabled)**
  Sets whether or not the tab at index is enabled. An internal exception is raised if there is no tab at that index.v<

- **public int getSelectedIndex()**

  Returns the currently selected index for this tabbedpane. Returns -1 if there is no currently selected tab.

1.6.1.5 JOptionPane

JOptionPane makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something. Almost all uses of this class are one-line calls to one of the static showXxxDialog methods shown below:

showConfirmDialog Asks a confirming question, like yes/no/cancel.
showInputDialog Prompt for some input.
showMessageDialog Tell the user about something that has happened.
ShowOptionDialog The Grand Unification of the above three.

The commonly used method is showMessageDialog.

- **public static void showMessageDialog(Component parentComponent, Object message, String title, int messageType) throws HeadlessException**

  Brings up a dialog that displays a message using a default icon determined by the messageType parameter.

  Parameters:

  - parentComponent - determines the Frame in which the dialog is displayed; if null, or if the parentComponent has no Frame, a default Frame is used
  - message - the Object to display
  - title - the title string for the dialog
  - messageType - the type of message to be displayed: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, or PLAIN_MESSAGE

1.6.2 Label

A label is a Component that displays a single line of static text. It is useful for putting a title or message next to another component. The text can be centered or justified to the left or right.

Methods:

- **public Label (String label)**
  This constructor creates a Label whose initial text is label. By default, the label's text is left justified.
- **public void setBackground (Color c)**
  The setBackground() method changes the current background color of the area of the screen occupied by the component to c. After changing the color, it is necessary for the screen to refresh before the change has any affect. To refresh the screen, repaint() is called.
- **public synchronized void setFont (Font f)**
  The setFont() method changes the component's font to f. If the font family (such as TimesRoman) provided within f is not available on the current platform, the system uses a default font family, along with the supplied size and style (plain, bold, italic). Depending upon the platform, it may be necessary to refresh the component/screen before seeing any changes.

- **public void setForeground (Color c)**
  The setForeground() method changes the current foreground color of the area of the screen occupied by the component to c. After changing the color, it is necessary for the screen to refresh before the change has any effect. To refresh the screen, call repaint().

1.6.3 Button

The Button component provides one of the most frequently used objects in graphical applications. When the user selects a button, it signals the program that something needs to be done by sending an action event. The program responds in its actionPerformed() method.

Methods:

- **public Button (String label)**

  This constructor creates a Button whose initial text is label.

1.6.4 TextField

TextField is the TextComponent for single-line input. Some constructors permit you to set the width of the TextField on the screen, but the current LayoutManager may change it. The text in the TextField is left justified, and the justification is not customizable.

Methods:

- **public TextField (int columns)**

  This constructor creates an empty TextField. The TextField width is columns. The TextField will try to be wide enough to display columns characters in the current font and size. As it is mentioned previously, the layout manager may change the size.
- **public String getText()**

  Returns the text contained in this TextComponent. If the underlying document is null, will give a NullPointerException.

1.6.5 ComboBox

A component that combines a button or editable field and a drop-down list. The user can select a value from the drop-down list, which appears at the user's request. If you make the combo box editable, then the combo box includes an editable field into which the user can type a value.

Methods:

- **public JComboBox()**
  Creates a JComboBox with a default data model. The default data model is an empty list of objects. Use addItem to add items. By default the first item in the data model becomes selected.
- **public void addItem(Object anObject)**
  Adds an item to the item list. This method works only if the JComboBox uses a mutable data model.
- **public Object getSelectedItem()**
  Returns the current selected item. As you can see, the method returns Object, so if you want to read a string from the combobox, you should first convert this Object to String with the toString() method. This method is inherited from Component Class.
  - **public String toString()**
    Returns a string representation of this component and its values.

```
String gauges_no= (comboBox1.getSelectedItem()).toString();
```

1.6.6 TextArea

TextArea is the TextComponent for multiline input. Some constructors permit you to set the rows and columns of the TextArea on the screen. However, the LayoutManager may change your settings. The text in a TextArea appears left justified, and the justification is not customizable.

Methods:

- **public TextArea (int rows, int columns)**
  This constructor creates an empty TextArea with both scrollbars. The TextArea is rows high and columns wide.
- **public void append (String string)**
  The append() method inserts string at the end of the TextArea.

1.7 Event-Driven Programming

Java is an event-driven environment, meaning that most actions that take place in Java generate an event that can be handled and responded to. In Java, an event is defined quite literally as something that happens that you might want to know about.

In the event-driven world of Java, the flow of your program follows events external to your application, as opposed to following an internally linear program flow. This is an important point, because it means that a Java application is in a constant state of responding to events with provided methods, which are called event handlers.

Because of the inherent graphical nature of Java applets, it will eventually become obvious to you why the event-driven programming model is not only more convenient, but downright necessary. With the potential of having multiple frames on a single application, along with on-the-fly system configuration changes and a multitude of other things going on, like drawing graphics as in my project, a procedural programming model would be much more difficult to manage. The event-based model provides a more sound solution to the problems inherent in a system with a graphical interface, such as Java.

All events in Java are processed within the java.awt (Advanced Windowing Toolkit) package, and are tightly linked to AWT components. As it is said before, Frames are themselves a specific type of component. This means that they inherit the same event-processing features built in to the Component superclass.

1.7.1 AWT Event Handling

The Java AWT is responsible for generating events in response to user actions. For example, when the user selects a button, an event of type ACTION_EVENT is generated. These events are in turn processed by applications, who use the AWT to respond to the events in an event-driven manner.

Somewhere deep inside the AWT is an event-processing loop, which handles the dirty job of routing events to their appropriate targets. This process of routing an event to a target object is known as posting an event. For target objects derived from Component, postEvent will in turn call the handleEvent method. handleEvent serves as the default handler for all events, and it has the option of responding to an event or letting it pass through. If handleEvent doesn't handle an event, it returns false, in which case the parent object's handleEvent method is called. This process continues until an event is handled or the top of the object tree is reached.

The Java AWT provides a class for encapsulating all types of events that can occur within the system: Event. The Event class models a generic event and has constants defined within it to represent specific events. The Event class is used primarily by the handleEvent method. handleEvent takes an Event object as its only parameter. handleEvent uses this Event object to determine what type of event has occurred. It then calls a more specific event-handler method to deal with the specific event. For example, if a key is pressed, the Event object's id member variable is set to KEY_PRESS, which is a constant defining the key press event. handleEvent checks the value of id and upon finding it equal to KEY_PRESS, calls the keyDown handler method.

1.7.2 Identifying the Target

All events occur within a Java Component. The program decides which component gets the event by starting at the outermost level and working in. For example user clicks a button. This action results in a call to the Frame's deliverEvent() method, which determines which component within the frame should receive the event and calls that component's deliverEvent() method.

1.7.3 Dealing With Events

Once deliverEvent() identifies a target, it calls that target's handleEvent() method (in this case, the handleEvent() method of Button) to deliver the event for processing. If Button has not overridden handleEvent(), its default implementation would call Button's action() method. If Button has not overridden action(), its default implementation (which is inherited from Component) is executed and does nothing. For your program to respond to the event, you would have to provide your own implementation of action() or handleEvent().

handleEvent() plays a particularly important role in the overall scheme. It is really a dispatcher, which looks at the type of event and calls an appropriate method to do the actual work: action() for action events, mouseUp() for mouse up events, and so on.

1.7.4 Listeners and Adapters

An event is propagated from a source object (component) to a "listener" object. The source object is the GUI object. Listeners are objects that register themselves as interested in certain kinds of events on that source. Listeners can be other GUI objects or separate objects responsible for delegated events. In the AWT, events that are passed are typically derived from the AWTEvent class.

The AWT provides for two conceptually different kinds of events: low-level and semantic. Low-level events are concerned with the specific user input or window system-level event occurrences. Semantic events are concerned not with the specifics of what interaction occurred, but with its meaning. For example, when you click a button or double-click an item in a list, you want to perform an action. However, if you click a scrollbar (or drag its marker), you want to adjust the value of something. You can see that the low-level event (the click with the mouse) does not perform the same role for different components; this is why you need semantic events.

Commonly used low-level events are :

- WindowEvent, that deal with the actions concern with the frame.
- PaintEvent, which is normally used only internally; it is not designed to be used with the listener model.

Commonly used semantic events are :

- ActionEvent encapsulates our notion of "doing something" or "performing an action."
- ItemEvent indicates that the state of an item has changed.

According to this model, all your class has to do is to implement a certain listener interface and get itself registered with the source object. However, listeners for low-level events are designed to listen to multiple event types (the WindowListener, for example, listens to window activation, closed, closing, deactivation, deiconification, iconification, and opened events); if your class implements such an interface, you must supply the methods that handle these events.

To make things simpler, the AWT event model includes adapter classes for low-level event listeners. These classes, located in the java.awt.event package, provide default implementations of all the methods so that you can choose which methods to override in your code. Because the body of the adapter method is empty, using adapters is equivalent to using the listener interfaces directly except that you don't have to specify every method in the listener interface.

For windowClosing event this technique is used in the project;

```
this.addWindowListener(new WindowAdapter()
      {     public void windowClosing(WindowEvent event)
                  {     shutDown(); }
      }     );
```

1.8 Exception Handling

1.8.1 What Is Exception Handling?

The Java programming environment borrows a very powerful error-handling technique from C++ known as exception handling. An exception is defined as an abnormal event that disrupts the normal flow of a program. Exception handling, therefore, is the process of detecting and responding to exceptions in a consistent and reliable manner. The term "exception" is used instead of "error" because exceptions represent exceptional, or abnormal, conditions that aren't necessarily errors. In this way, an exception is a very general concept, meaning that any number of abnormal events could be interpreted as an exception.

1.8.2 Throwing Exceptions

Any code that is capable of throwing an exception must specifically be designed to do so. In other words, exceptions aren't just something that magically appear whenever a problem occurs; code that has the potential of causing problems must be designed to notify a program of these problems accordingly. The Java throws keyword provides the necessary mechanism to wire exception information into code that is potentially dangerous. The throws keyword is used at the method level, meaning that you declare a particular method capable of throwing a certain exception or set of exceptions, like this:

```
public void thisIsTrouble() throws anException
        { // method body }
```

In this example, the throws keyword is used to specify that the thisIsTrouble method can generate an exception of type anException. Any code that calls this method knows immediately that the risk is there for the method to generate an anException exception.

1.8.3 Catching Exceptions

Basically, when you need to watch out for an exception, you enclose the code that may generate the error in a try program block. If the code in the block generates an exception, you handle that exception in a catch program block.

```
try { thisIsTrouble(); }
catch (anException e) { System.out.println(e.getMessage()); }
```

This code makes a call to the thisIsTrouble method inside a try block. A try block is used to hold code that is at risk of throwing an exception. Any code executing in a try block is considered at risk of throwing an exception. If an exception occurs in the try block, the runtime system looks to the catch clause to see if the exception type matches the one that was thrown. If so, the code in the catch block is executed.

CHAPTER 3
UML : UNIFIED MODELING LANGUAGE

3.1 What is The Unified Modeling Language?

Some languages, such as Java, require an object-oriented structure. The object-oriented paradigm is a different way of viewing applications. With object-oriented programming, developers create blocks of code, called objects. You can then build the application by piecing all of these objects together.

The Unified Modeling Language (UML) is the standard graphical language for specifying the analysis and design of object-oriented software. The UML defines a set of diagrams, also called models, with well-defined graphical symbols so developers can easily understand each other's diagrams. The graphical symbols are augmented by textual descriptions that also aid in model comprehension.

You cannot look at a UML diagram and say exactly what the equivalent code would look like. However, you can get a rough idea of what the code would look like. In practice, that's enough to be useful. Development teams often form their local conventions for these, and you'll need to be familiar with the ones in use.

3.2 Fundamental Concepts

Classes and objects are two fundamental concepts in UML. A class is an abstraction of a set of real-world things that all have the same data and behavior. A class may have many instances called objects. Each object conforms to the rules defined by its class.

Classes cooperate with one another to satisfy the requirements of the system. Associations link classes together. Classes communicate with one another by passing events and invoking services.

3.3 Visual Modeling

Visual modeling is the process of taking the information from the model and displaying it graphically using a standard set of graphical elements. A standard is important to realizing one of the benefits of visual modeling: communication. Communication between users, developers, analysts, testers, managers, and anyone else involved with a project is the primary purpose of visual modeling. You could accomplish this communication using non-visual (textual) information, but on the whole, humans are visual creatures. We seem to be able to understand complexity better when it is displayed to us visually as opposed to written textually. By producing visual models of a system, we can show how the system works on several levels. We can model the interactions between the users and a system. We can model the interactions of objects within a system. We can even model the interactions between systems, if we so desire.

After creating these models, we can show them to all interested parties, and those parties can glean the information they find valuable from the model. For example, users can visualize the interactions they will make with the system from looking at a model. Analysts can visualize the interactions between objects from the models. Developers can visualize the objects that need to be developed and what each one needs to accomplish. Testers can visualize the interactions between objects and prepare test cases based on these interactions. Project managers can see the whole system and how the parts interact. And chief information officers can look at high-level models and see how systems in their organization interact with one another. All in all, visual models provide a powerful tool for showing the proposed system to all of the interested parties.

3.4 Systems of Graphical Notation

One important consideration in visual modeling is what graphical notation to use to represent various aspects of a system. This notation needs to be conveyed to all interested parties or the model will not be very useful. Many people have proposed notations for visual modeling. Some of the popular notations that have strong support are both, Object Modeling Technology (OMT), and UML.

Rational Rose supports these three notations; however, UML is a standard that has been adopted by the majority of the industry as well as the standards' governing boards such as ANSI and the Object Management Group (OMG).

3.5 Understanding UML Diagrams

UML allows people to develop several different types of visual diagrams that represent various aspects of the system. Rational Rose supports the development of the majority of these models, as follows:

- Business Use Case diagram
- Use Case diagram
- Activity diagram
- Sequence diagram
- Collaboration diagram
- Class diagram
- Statechart diagram
- Component diagram
- Deployment diagram

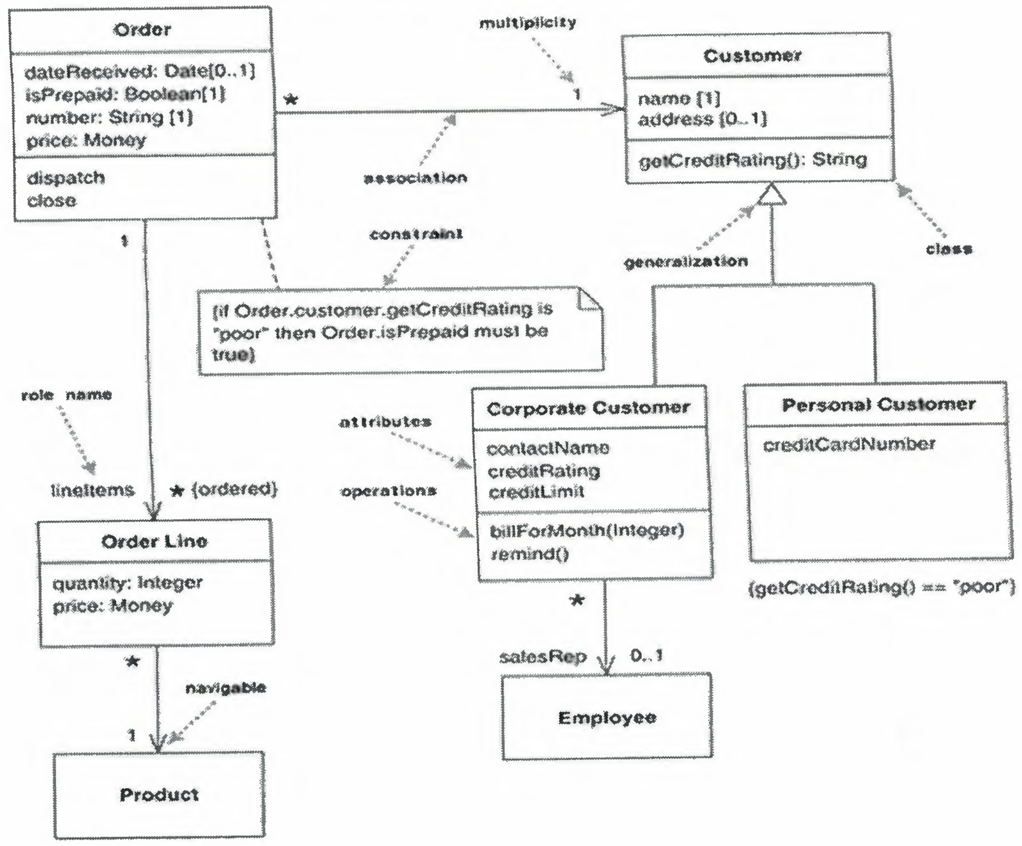
These model diagrams illustrate different aspects of the system. For example, the Collaboration diagram shows the required interaction between the objects in order to perform some functionality of the system. Each diagram has a purpose and an intended audience. The diagrams used in this project are;

3.5.1 Class Diagrams

A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. The UML uses the term feature as a general term that covers properties and operations of a class.

Figure 3.1 shows a simple class model. The boxes in the diagram are classes, which are divided into three compartments: the name of the class (in bold), its attributes, and its operations. Figure 3.1 also shows two kinds of relationships between classes: associations and generalizations.

Figure 3.1. A simple class diagram



3.5.1.1 Properties

Properties represent structural features of a class. Properties are a single concept, but they appear in two quite distinct notations: attributes and associations. Although they look quite different on a diagram, they are really the same thing especially for Java. Because actually associations are simply established by defining attributes, that are type of other classes, in Java.

3.5.1.2Attributes

The attribute notation describes a property as a line of text within the class box itself. The full form of an attribute is:

visibility name: type multiplicity = default {property-string}
An example of this is:
- name: String [1] = "Untitled" {readOnly}

- This visibility marker indicates whether the attribute is public (+) or private (-).
- The name of the attribute—how the class refers to the attribute—roughly corresponds to the name of a field in a programming language.
- The type of the attribute indicates a restriction on what kind of object may be placed in the attribute. You can think of this as the type of a field in a programming language.
- The default value is the value for a newly created object if the attribute isn't specified during creation.
- The {property-string} allows you to indicate additional properties for the attribute. In the example, {readOnly} used to indicate that clients may not modify the property. If this is missing, you can usually assume that the attribute is modifiable.

3.5.1.3 Associations

The other way to notate a property is as an association. Much of the same information that you can show on an attribute appears on an association. Figures 3.2 and 3.3 show the same properties represented in the two different notations.

Figure 3.2. Showing properties of an order as attributes

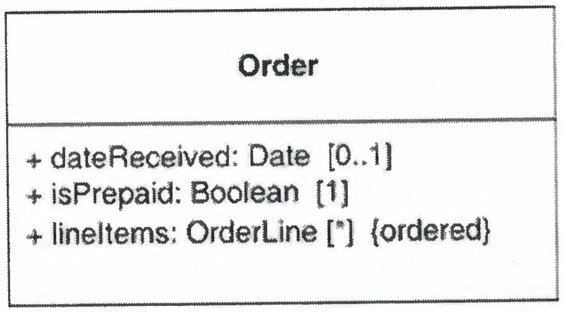
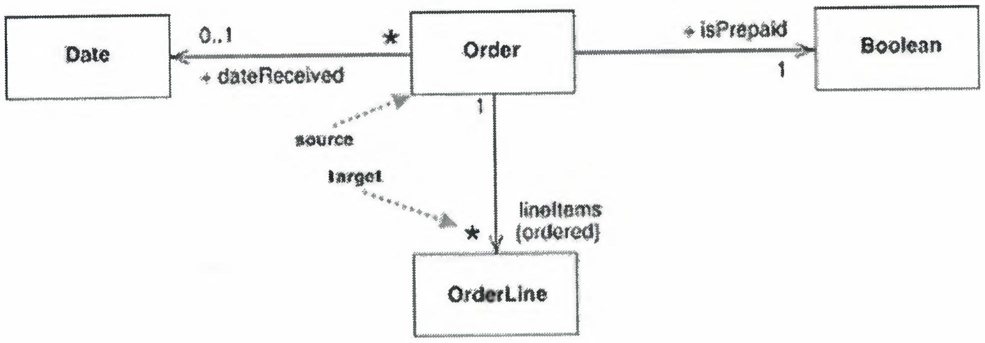

Order
+ dateReceived: Date [0..1]
+ isPrepaid: Boolean [1]
+ lineItems: OrderLine [*] {ordered}

Figure 3.3. Showing properties of an order as associations



An association is a solid line between two classes, directed from the source class to the target class. The name of the property goes at the target end of the association, together with its multiplicity. The target end of the association links to the class that is the type of the property.

Although most of the same information appears in both notations, some items are different. In particular, associations can show multiplicities at both ends of the line.

In general attributes are used for small things, such as integer or boolean and association for more significant user defined classes, such as General or Gauges. And also class boxes are used for classes that are significant for the diagram, which leads to using associations, and attributes for things less important for that diagram.

3.5.1.3.1 Multiplicity

The multiplicity of a property is an indication of how many objects may fill the property. The most common multiplicities are

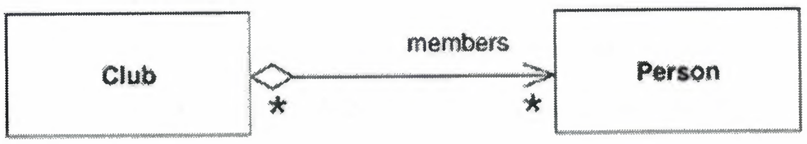
- 1 (An order must have exactly one customer.)
- 0..1 (A corporate customer may or may not have a single sales rep.)
- * (A customer need not place an Order and there is no upper limit to the number of Orders a Customer may place—zero or more orders.)

More generally, multiplicities are defined with a lower bound and an upper bound, such as 1..6 for the thread going to be used in the experiment.

3.5.1.4 Aggregation and Composition

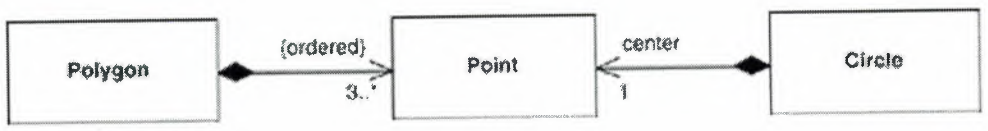
One of the most frequent sources of confusion in the UML is aggregation and composition. It's easy to explain glibly: Aggregation is the part-of relationship. It's like saying that a car has an engine and wheels as its parts. The difficult thing is considering what the difference is between aggregation and association.

Figure 3.4. Aggregation



As well as aggregation, the UML has the more defined property of composition. In Figure 5.4, an instance of Point may be part of a polygon or may be the center of a circle, but it cannot be both. The general rule is that, although a class may be a component of many other classes, any instance must be a component of only one owner. The class diagram may show multiple classes of potential owners, but any instance has only a single object as its owner.

Figure 3.5. Composition



The "no sharing" rule is the key to composition. Another assumption is that if you delete the polygon, it should automatically ensure that any owned Points also are deleted.

3.5.1.5 Operations

Operations are the actions that a class knows to carry out. Operations most obviously correspond to the methods on a class. Normally, you don't show those operations that simply manipulate properties, because they can usually be inferred.

The full UML syntax for operations is:
visibility name (parameter-list) : return-type {property-string}

- This visibility marker is public (+) or private (-).
- The name is a string.
- The parameter-list is the list of parameters for the operation.
- The return-type is the type of the returned value, if there is one.
- The property-string indicates property values that apply to the given operation.

The parameters in the parameter list are notated in a similar way to attributes. The form is:
direction name: type = default value

- The name, type, and default value are the same as for attributes.
- The direction indicates whether the parameter is input (in), output (out) or both (inout). If no direction is shown, it's assumed to be in.

An example operation on account might be:

+ balanceOn (date: Date) : Money

3.5.2 Sequence Diagrams

Interaction diagrams describe how groups of objects collaborate in some behavior. The UML defines several forms of interaction diagram, of which the most common is the sequence diagram.

Typically, a sequence diagram captures the behavior of a single scenario. The diagram shows a number of example objects and the messages that are passed between these objects within the use case.

Figure 3.6. A sequence diagram for centralized control

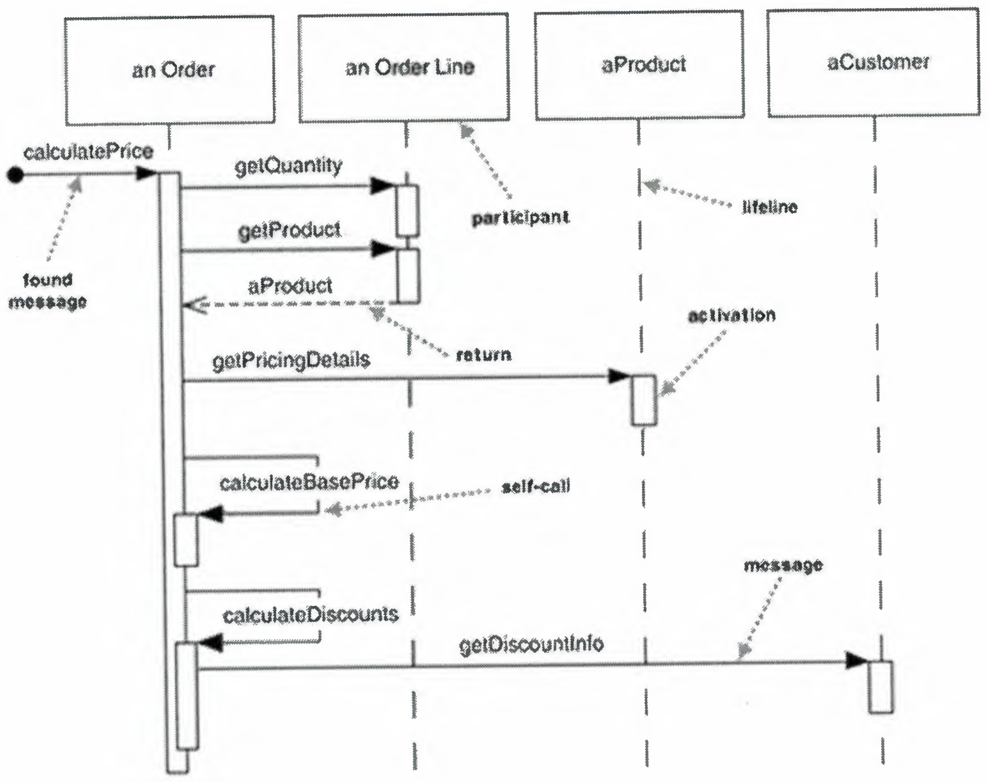


Figure 3.4 a sequence diagram that shows one implementation of a scenario. Sequence diagrams show the interaction by showing each participant with a lifeline that runs vertically down the page and the ordering of messages by reading down the page.
Sequence diagram indicates the differences in how the participants interact. This is the great strength of interaction diagrams. They aren't good at showing details of algorithms, such as loops and conditional behavior, but they make the calls between participants crystal clear and give a really good picture about which participants are doing which processing.

3.5.2.1 Synchronous and Asynchronous Calls

If you're exceptionally alert, you'll have noticed that the arrowheads in the last couple of diagrams are different from the arrowheads earlier on. That minor difference is quite important in UML 2. In UML 2, filled arrowheads show a synchronous message, while stick arrowheads show an asynchronous message.

If a caller sends a synchronous message, it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and doesn't have to wait for a response. You see asynchronous calls in multithreaded applications and in message-oriented middleware. Asynchrony gives better responsiveness and reduces the temporal coupling but is harder to debug.

CHAPTER 2
THREADS

A thread is an independent sequence of execution within a Java application. Every Java application runs within a Java VM (virtual machine). The Java VM may be, simultaneously, running multiple parts of a single application. Every Java application is given at least one thread and may create more at its discretion.

2.1 Thread Class

The Thread class is the primary class responsible for providing thread functionality to other classes. To add thread functionality to a class, you simply derive the class from Thread and override the run method. The run method is where the processing for a thread takes place, and it is often referred to as the thread body. The Thread class also defines start and stop methods that allow you to start and stop the execution of the thread, along with a host of other useful methods.

2.2 Runnable

Java does not directly support multiple inheritance, which involves deriving a class from multiple parent classes. This brings up a pretty big question in regard to adding thread functionality to a class: how can you derive from the Thread class if you are already deriving from another class? The answer is: you can't! This is where the Runnable interface comes into play.

The Runnable interface provides the overhead for adding thread functionality to a class simply by implementing the interface, rather than deriving from Thread. Classes that implement the Runnable interface simply provide a run method that is executed by an associated thread object that is created separately. This is a very useful feature and is often the only outlet you have to incorporating multithreading into existing classes

2.3 Object

Although not strictly a thread support class, the Object class does provide a few methods that are crucial to the Java thread architecture. These methods are wait, notify, and notifyAll. The wait method causes a thread to wait in a sleep state until it is notified to continue. Likewise, the notify method informs a waiting thread to continue along with its processing. The notifyAll method is similar to notify except it applies to all waiting threads. These three methods can only be called from a synchronized method.

Typically, these methods are used with multiple threads, where one method waits for another to finish some processing before it can continue. The first thread waits for the other thread to notify it so it can continue. Just in case you're in the dark here, the Object class rests at the top of the Java class hierarchy, meaning that it is the parent of all classes. In other words, every Java class inherits the functionality provided by Object, including the wait, notify, and notifyAll methods.

2.4 Creating Threads

In Java, you can create threads in a couple of ways. The first one is to take an existing class and turn it into a thread. You do this by modifying the class so that it implements the Runnable interface, which declares the run() method required by all types of threads. (The run()method contains the code to be executed by a thread.)

The second way to create a thread is to write a completely separate class derived from Java's Thread class. Because the Thread class itself implements the Runnable interface, it already contains a run() method. However, Thread's run() method doesn't do anything. You usually have to override the method in your own class in order to create the type of thread you want.

2.4.1 Converting a Class to a Thread

As it is mentioned before, the first way to create a thread is to convert a class to a thread. To do this, you must perform several steps, as listed here:

- Declare the class as implementing the Runnable interface.
- Implement the run() method.
- Declare a Thread object as a data field of the class.
- Create the Thread object and call its start() method.
- Call the thread's stop() method to destroy the thread.

2.4.1.1 Declaring the Class as Implementing the Runnable Interface

To create a thread from a regular class, the class must first be declared as implementing the Runnable interface. For example, if your class is declared as

```
public class MyApplet extends Applet
```

you must change that declaration to

```
public class MyApplet extends Applet  implements Runnable
```

2.4.1.2 Implementing the run() Method

Now, because you've told Java you're about to implement an interface, you must implement every method in the interface. In the case of Runnable, that's easy because there's only one method, run(), the basic implementation of which looks like this:

```
public void run()
   {      }
```

When you start your new thread, Java calls the thread's run() method, so it is in run() where all the action takes place.

2.4.1.3 Declaring a Thread Object

The next step is to declare a Thread object as a data field of the class, like this:

```
Thread thread;
```

The thread object will hold a reference to the thread with which the application is associated. You will be able to access the thread's methods through this object.

2.4.1.4 Creating and Starting the Thread Object

Now it's time to write the code that creates the thread and gets it going.

```
public static void main(String args[] )
 {   thread = new Thread(this);
     thread.start();
 }
```

Right after the call to the constructor, the applet calls the Thread object's start() method, which starts the thread running. When the thread starts running, Java calls the thread's run() method, where the thread's work gets done.

2.4.2 Deriving a Class from Thread

The second way to create a thread is to derive a new class from Thread. Then, in your application class, you create and start a thread object of your thread class. This leaves you with two processes going simultaneously, the application and the thread object created in the class. By giving the thread class access to data and methods in the application, the thread can easily communicate with the application in order to perform whatever tasks it was written for.

```
public class MyThread extends Thread
  { MainClass main;
    MyThread ( MainClass main )
```

```
        { this.main = main;  }
    }

    Public class MainClass extends JFrame
    {  MyThread thread;
       Public static void main ( String args[]  )
         { thread=new MyThread ( this );  }
    }
```

As seen there are two classes here. One extends to Thread and the other extends to JFarme. The Constructor of MyThread class take an MainClass typed argument and assigns this argument to its local variable which is also typed as MainClass. By means of this local variable The MyThread class can access public variables and methods of MainClass class. While MainClass calling the constructor of MyThread class, it sends this object, which is actually the class itself, as an argument.

2.5 Thread Attributes

Each thread contains a number of attributes. Some must be set when the thread object is created and can never be altered, while others can be changed throughout the thread's life. The attributes are: name, priority and thread group.

- The thread group must be specified while creating the thread and cannot change.
- The name can be queried and set using the getName() and setName() methods.
- The priority defaults to the priority of the creating thread. The current priority can be obtained from the getPriority() method. Before starting a thread and during its execution the threads priority can be altered by calling the setPriority() method with the desired priority. The priority can not be set higher than the maximum priority of the thread's group. If an attempt is made to set the priority higher it will silently be

ignored and the priority will be changed to be equivalent to the current maximum priority of the threads group.

2.6 Multithreading

Multithreaded programs are similar to the single-threaded programs. They differ only in the fact that they support more than one concurrent thread of execution-that is, they are able to simultaneously execute multiple sequences of instructions. Each instruction sequence has its own unique flow of control that is independent of all others.

In single-processor systems, only a single thread of execution occurs at a given instant. The CPU quickly switches back and forth between several threads to create the illusion that the threads are executing at the same time.

The advantage of multithreading is that concurrency can be used within a process to provide multiple simultaneous services to the user. Multithreading also requires less processing overhead than multiprogramming because concurrent threads are able to share common resources more easily. Multiple executing programs tend to duplicate resources and share data as the result of more time-consuming interprocess communication.

2.6.1 Thread States

As it is mentioned before a thread is created by creating a new object of class Thread or of one of its subclasses. When a thread is first created, it does not exist as an independently executing set of instructions. Instead, it is a template from which an executing thread will be created. It first executes as a thread when it is started using the start() method and run via the run() method. Before a thread is started it is said to be in the new thread state. After a thread is started, it is in the runnable state. When a class is in the runnable state, it may be executing or temporarily waiting to share processing resources with other threads. A runnable thread enters an extended wait state when one of its methods is invoked that causes it to drop from the runnable state into a not runnable state. In the not runnable state, a thread is not just waiting for its share of processing resources, but is blocked waiting for the occurrence of an event that will send it back to the runnable state.

For example, the sleep() method was invoked in the ThreadTest1 and ThreadTest2 programs to cause a thread to wait for a short period of time so that the other thread could execute. The sleep() method causes a thread to enter the not runnable state until the specified time has expired.

A thread leaves the not runnable state and returns to the runnable state when the event that it is waiting for has occurred. For example, a sleeping thread must wait for its specified sleep time to occur.

A thread may transition from the new thread, runnable, or not runnable state to the dead state when its stop() method is invoked or the thread's execution is completed. When a thread enters the dead state, it's a goner. It can't be revived and returned to any other state.

2.6.2 Thread Priority and Scheduling

From an abstract or a logical perspective, multiple threads execute as concurrent sequences of instructions. This may be physically true for multiprocessor systems, under certain conditions. However, in the general case, multiple threads do not always physically execute at the same time. Instead, the threads share execution time with each other based on the availability of the system's CPU (or CPUs).

The approach used to determining which threads should execute at a given time is referred to as scheduling. Scheduling is performed by the Java runtime system. It schedules threads based on their priority. The highest-priority thread that is in the runnable state is the thread that is run at any given instant. The highest-priority thread continues to run until it enters the death state, enters the not runnable state, or has its priority lowered, or when a higher-priority thread becomes runnable.

A thread's priority is an integer value between MIN_PRIORITY and MAX_PRIORITY. These constants are defined in the Thread class. MIN_PRIORITY is 1 and MAX_PRIORITY is 10. A thread's priority is set when it is created. It is set to the same priority as the thread that created it. The default priority of a thread is NORM_PRIORITY and is equal to 5. The priority of a thread can be changed using the setPriority() method.

Java's approach to scheduling is referred to as preemptive scheduling. When a thread of higher priority becomes runnable, it preempts threads of lower priority and is immediately executed in their place. If two or more higher-priority threads become runnable, the Java scheduler alternates between them when allocating execution time.

2.6.3 Synchronization

There are many situations in which multiple threads must share access to common objects. For example, in my project several threads calculate some values and want to draw a graph to the screen. Java enables you to coordinate the actions of multiple threads using synchronized methods and synchronized statements.

An object for which access is to be coordinated is accessed through the use of synchronized methods. These methods are declared with the synchronized keyword. Only one synchronized method can be invoked for an object at a given point in time. This keeps synchronized methods in multiple threads from conflicting with each other.

All classes and objects are associated with a unique monitor. The monitor is used to control the way in which synchronized methods are allowed to access the class or object. When a synchronized method is invoked for a given object, it is said to acquire the monitor for that object. No other synchronized method may be invoked for that object until the monitor is released. A monitor is automatically released when the method completes its execution and returns. A monitor may also be released when a synchronized method executes certain methods, such as wait(). The thread associated with the currently executing synchronized method becomes not runnable until the wait condition is satisfied and no other method has acquired the object's monitor.

The following example shows how synchronized methods and object monitors are used to coordinate access to a common object by multiple threads.

```
class ThreadSynchronization
 { public static void main(String args[])
    { MyThread thread1 = new MyThread("thread1: ");
      MyThread thread2 = new MyThread("thread2: ");
      thread1.start();
      thread2.start();
    }
 }
class MyThread extends Thread
 { public MyThread(String id)
     { super(id);  }
 public void run()
   { SynchronizedOutput.displayList(getName(),message); }
 void randomWait()
   { try {  sleep((long)(3000*Math.random())); }
     catch (InterruptedException x){    }
   }
}
class SynchronizedOutput
  { public static synchronized void displayList(String name,String list[])
      { MyThread t = (MyThread) Thread.currentThread();
        t.randomWait();
        System.out.println(name+list[i]);
        }
    }
```

It is important to understand that the displayList() method is static and applies to the SynchronizedOutput class as a whole, not to any particular instance of the class. It invokes randomWait() to wait a random amount of time before operating. The displayList() method uses the currentThread() method of class Thread to reference the current thread in order to invoke randomWait().

When displayList() is not synchronized, it may be invoked by one thread, say thread1, display some output, and wait while thread2 executes. When thread2 executes, it too invokes displayList() to display some output. Two separate invocations of displayList(), one for thread1 and the other for thread2, execute concurrently. This results with mixed output display.

When the synchronized keyword is used, thread1 invokes displayList(), acquires a monitor for the SynchronizedOutput class (because displayList() is a static method), and displayList() proceeds with the output display for thread1. Because thread1 acquired a monitor for the SynchronizedOutput class, thread2 must wait until the monitor is released before it is able to invoke displayList() to display its output. This lets one task's output completed before the other's.

CHAPTER 4
SOFTWARE DESIGN FOR MONITORING THE PARAMETER OF ENVIRONMENT : JAVA APPLICATION

4.1 Software Development Process

4.1.1 Requirement Capture and Analysis

The purpose of the requirement capture analysis is to aim the development toward the right system. Its goal is to produce a document called requirement specification.

The document of the requirement specification will be used as

1. The agreed contract between the client and the system development organization on what the system should do (and what the system should not do);
2. The basis used by the developer to develop the system;
3. A fairly full model of what is required of the system..

4.1.1.1 The Requirement Specification of Wind – tunnel Monitor Software

1. The program is to monitor pressure and temperature from strain gauges placed on objects held in a wind tunnel.
2. The program must be written in C++ or Java Programming Languages on the computer IBM PC.
3. Signals received by the computer will come from a serial line. Each signal will be in ASCII format and will represent a pressure and temperature reading for 1 second.
4. The program will monitor the pressure and temperature reading over a period of time no longer than 1 hour.
5. A general aim in the design of the program is to minimize the amount of main memory used.

6. Before a wind-tunnel experiment starts the tunnel operator must provide the program with the following items of data:
   - A symbolic name for monitoring point and the amount of the pressure and temperature gauges (equal to each other) associated with that point. Maximum amount is less than or equal to 5.
   - The duration of the experiment (not longer than 1 hour).
   - The period over which a pressure average and temperature average is to be taken (AVERAGEPERIOD).
7. At the end of the wind-tunnel experiment the program must produce for each pressure and temperature gauges a report. This must consist of:
   - The name of the point.
   - A list of temperature and pressure values read in each seconds and averages over AVERAGEPERIOD for the duration of experiment.

4.1.1.2 Analyzing Requirements

In this step, the collected requirements should be analyzed and categorized. The requirements can be divided in to four categories.

1. Functional Requirements: Are the requirements that are directly related with the functionality of the software like reporting, displayed data, shortly the properties of the software.
2. Non-Functional Requirements: Are the requirements that are not related with the functionality of the software. These are mostly the client restrictions that must be satisfied. These can be memory usage restrictions, some standard restrictions or the restrictions about the size of the database that is doing to be used.
3. Design Objective: Are the requirements that are the main designing objectives of the software like easy and efficient maintenance.
4. Design Decision: Are the requirements that are related with the coding level of the development process. These requirements should be taken into consideration during writing the code.

Category	Req_no
Functional Requirements	1,4,6,7
Non-Functional Requirements	2
Design Objective Requirements	5
Design Decision Requirements	3

4.1.2 System Design

After analyzing the requirements, it is time to start designing the system. Non-Functional and Design Objective are the highest important requirements, that should be satisfied in the design. Use Java or C++ and minimize the memory usage. When concentrated on these, it can be seen that these two actually say the same thing: 'Use Object Oriented Architecture'.

4.1.2.1 Designing User Interface

The visual design of the software is done before the system design. Because in Object Oriented Programming Languages, especially in Java, the general design of the user interface also affects the system design much.

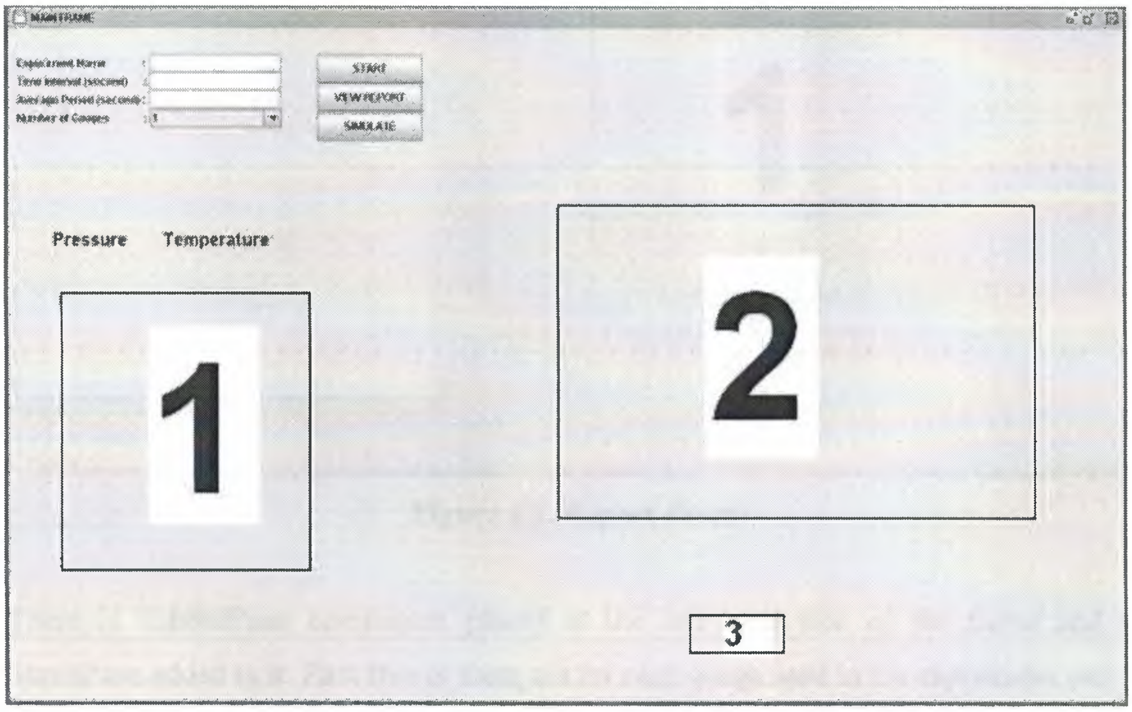


Figure 4.1. Main Frame

Figure 4.1. is the screen shot of the main frame of the application. There are 3 TextFields; one for the experiment name, other for the duration of the experiment and the other for the average period. A ComboBox to determine the number of gauge that is going to be used in the experiment. Three buttons to start the experiment, to view the report and to simulate the experiment. The red box with number 1 is the place where the instant temperature and pressure will be displayed. In the place of box number 2, the average graphic will be displayed. And box 3 is the place of clock that will show the time which average is taken.

When the report button pressed, a new frame will be opened which can be seen in Figure 4.2.

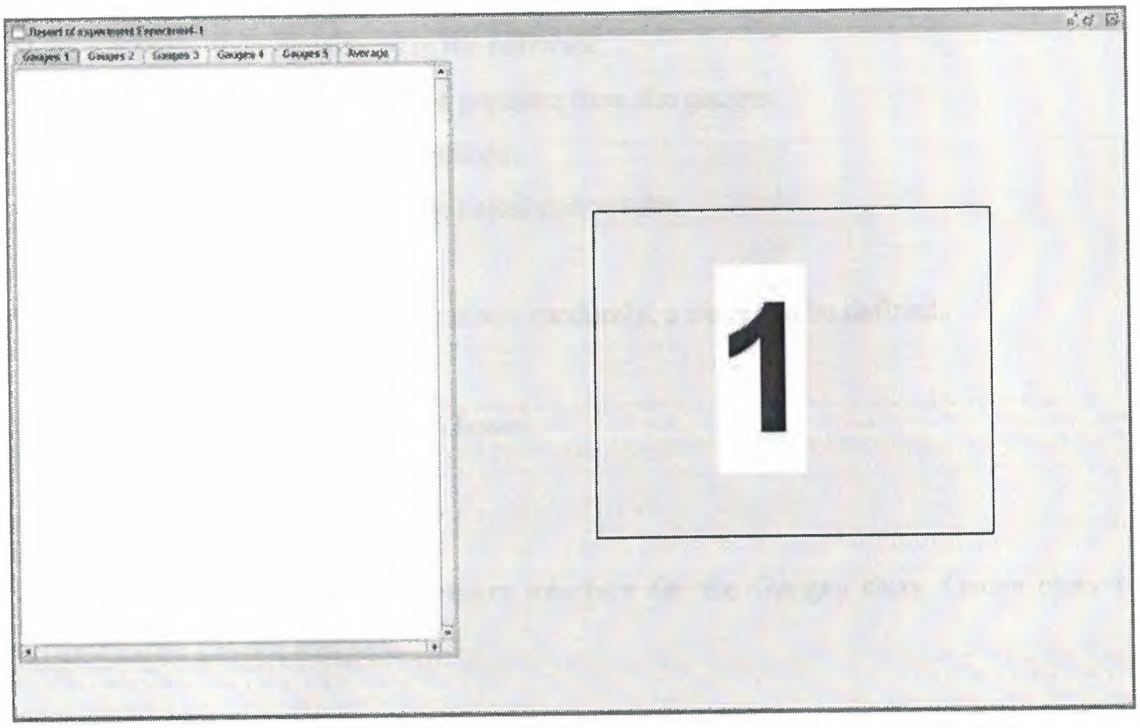


Figure 4.2. Report Frame

There is TabbedPane component placed at the left hand side of the frame and six ScrollPane added in it. First five of them are for each gauge used in the experiment and the last for average. The pressure and temperature measured at specific times will be registered to these ScrollPanes. In the place of box number 3, the graphic of the selected Tab will be drawn.

4.1.2.2 Structuring Collaboration Diagram

After designing the graphical user interface, some classes are cleared in the design. These are;

- Main_Class
- Report_Frame

Both these classes extends to JFrame class.

Now the operations should be analyzed deeply and catch other classes for the design. There are three main operations in the software;

1. Getting the temperature and pressure from the gauges.
2. Doing the necessary calculations.
3. Drawing the graphics of the calculated results.

For Getting the temperature and pressure randomly, a class can be defined.

- Randomizing

For calculations, there can be two classes.

- General
- Gauges

General class will provide a common interface for the Gauges class. Gauge class will accommodate general methods.

After deep analyze, the collaboration diagram can be shaped as in Figure 4.3.

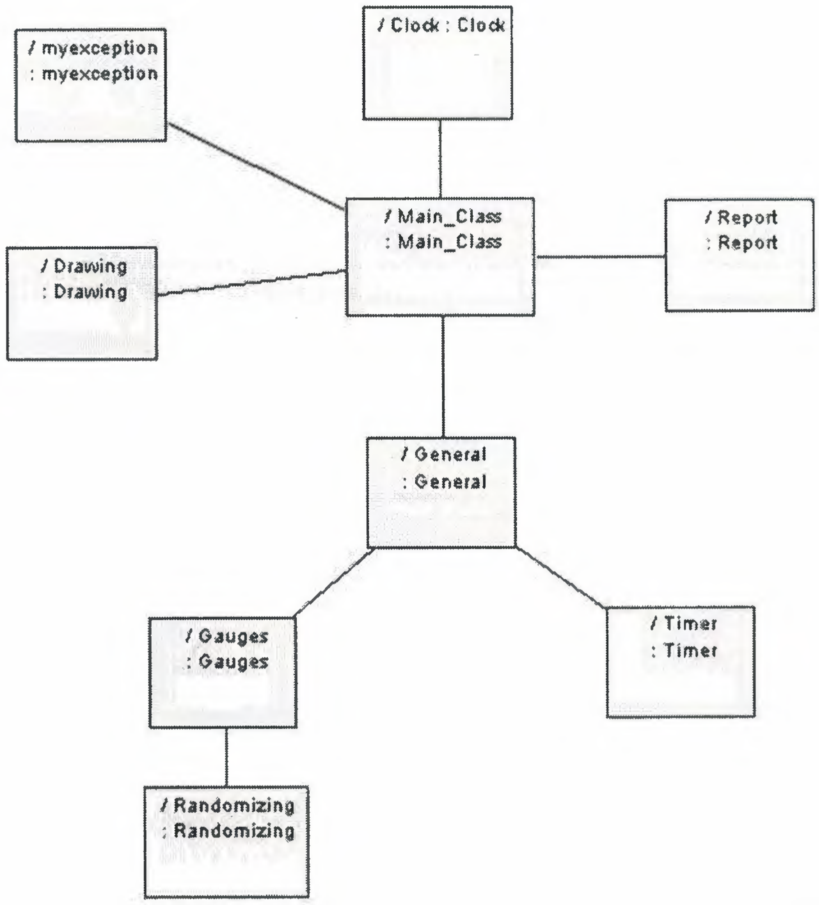


Figure 4.3. Collaboration Diagram

This diagram shows the general view of classes and their association between them. Although the diagram seems meaningless, it gives some idea for future design of the system.

4.1.2.3 Structuring Class Diagram

By analyzing the Collaboration Diagram deeply a well defined Class Diagram is build (Figure 4.4.)

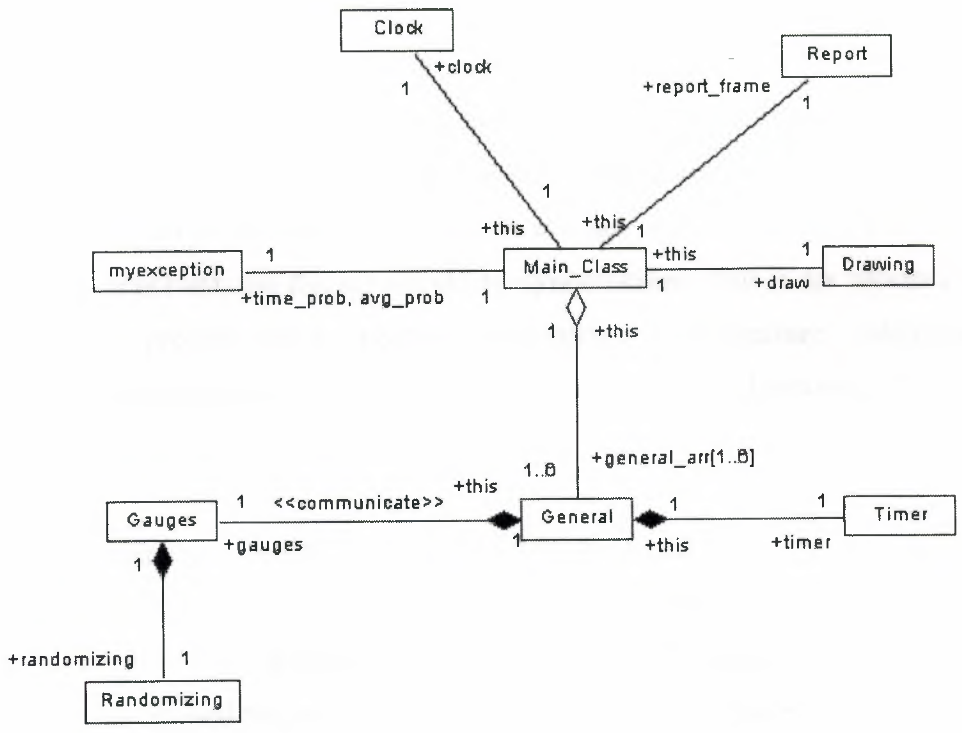


Figure 4.4. Class Diagram

This Class Diagram gives more detailed information about the structure of the design and associations between classes. Examining the diagram shows that, instances of the General class are created dynamically for each gauges object, that is going to be used in the experiment, and one more for average. Taking average of the gauges can also be considered as a gauge or lets say source. Just the data source is changing. Gauges take their data from Randomizing class and the Average takes its data from the other gauges.

Each created General class is the top class created for each gauges object and will be responsible from its assigned Gauges object. It will control reading temperature and pressure, calculate the average up to that moments starting from previous average period and calculate the new graphics coordinates. All these operations should be done just once in a second of time for each gauge. When average period comes, the instance of the General class created for the average does the same process with the difference of calculating average.

With this analyze, it is comprehensive that timing and continues flow of execution for each General objects is important. Then it can be said that the General class should be Thread. By extending the General class to the Java's Thread class, the continues flow of execution for General class can be satisfied, but not enough. As this is a real-time application and timing is important, the threads should be synchronized. This is for preventing the threads to execute process blocks (getting temperature and pressure, calculating graphical coordinates) just ones in a second. In other words, in a second just one value should be read for each gauge, not more than once.

With this purpose, each General class creates an instance of Timer class to it self. This Timer class will have some methods to get the owner thread sleep and notify. Initially, all threads are sleeping. Then the Main_Class sends a notify signal to the first thread to wake it up. As it mentioned the necessary code for sending sleep and notify signals are capsulated by the Timer class which's instances are created by each thread. After thread one completes the first iteration of the process blocks, it sends the notify signal to the second thread and a sleep signal to it self. So the first thread gets in to sleep until the last thread notifies it. This is repeated until the last thread. The last thread is for average. So it will work only when average period comes. In any condition the last thread waits for the second to flow before notifying the first thread. So after a second pass it can be seen that each thread read only one value for pressure and temperature.

As a principle of Object Oriented Architecture, the jobs should be distributed to several classes. So in the meaning of relieving the tasks of General class, logically separated tasks can be collected in different classes. With this purpose, reading values from the gauges and calculating the graphical coordinates can be distributed to two different classes. Randomizing class will generate random numbers for temperature and pressure. And Gauges class will do the calculations for graphics.

Most of the parts of the system design is cleared. The classes and their associations are designated. To go further, now the classes should be analyzed by one by and decisions should be made on attributes and methods on each class.

4.1.2.3.1 Class Analyze

4.1.2.3.1.1 Main_Class

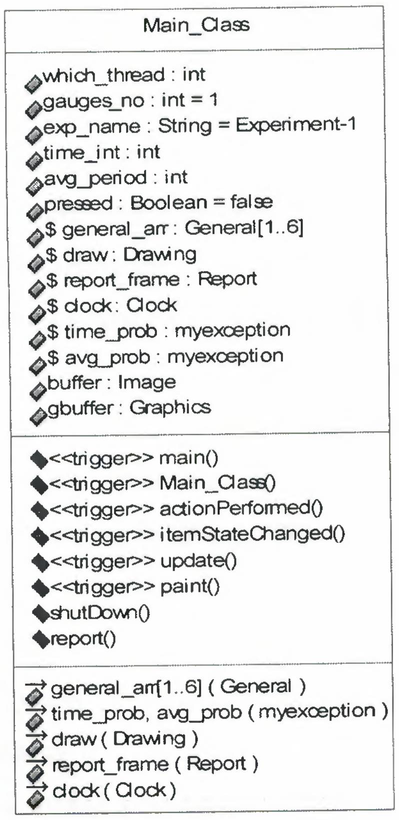


Figure 4.5. Main_Class class

Attributes :

- **static int which_thread**

  The paint( ) method, which is used to draw graphics on to screen, is an inherited method from Java's Component class. This means that you can use this method only if your class extends to a class which is a child of Component class like Frame. As there is only one paint( ) method and several threads want to draw something with this method, it should be known which thread calls the paint method. Which_thread attribute is used for this purpose. Threads set this attribute with their own id and then call the Main_Class's paint( ) method.

- **int gauges_no**
  Keeps the number of gauges will be used in the experiment. By adding one to this attribute, the number of threads will be found (extra one for average). Its default value is 1.
- **int avg_period**
  In how many seconds should the average be taken? The answer is in this attribute.
- **Image buffer & Graphics gbuffer**
  There are two ways of drawing a graphic on the screen. One is; draw the graph directly on the canvas by using the 'g' variable (which's type is Graphics and comes as a parameter from the system call and assigned to a default Image object) in the paint( ) method and the other is to define your own Graphics object and Image object, assign your Graphic object to your Image object by the getImage( ) method and use this Graphic object while drawing. This method is called off-screen method and these attributes are for this purpose.
- **General general_arr[]**
  This is an array to keep the dynamically created instances of General class. As it keeps the instance objects of General class, its type should be General as well. Keeping these instances in an array, gives chance to access these objects sequentially, which is needed as understood from the system design. Its size is 6.

Methods :

- **public static void main (String args[])**
  This method is essential for all Java Applications. When the application started, Java Run-Time Environment starts the execution from the main( ) method. This function is defined as Trigger in the UML Class diagram because actually it is triggered by the system.
- **public Main_Class( )**

This is the constructor of the class. When the Java Run_Time Environment starts the execution from the main( ), it should find a call to constructer of Main_Class. In the constructor, the global variables should be initiated and the appearance of the frame should be set. This is a Trigger method.

- **public void actionPerformed(ActionEvent e)**

  For catching events on some components the class should implements a Listener interface. Implementing an interface requires its all methods to be overridden by the implementing class. That is why, to catch the button events, Main_Class should implements to ActionListener interface class and must override the actionPerformed method of this interface class. There can be many components registered to the ActionListener to catch the events. When the ActionEvent accurse on one of these components, system automatically calls the actionPerformed( ) method to be executed and pass a parameter to the method, that defines which component triggered the event.

- **public void itemStateChanged (ItemEvent ie)**

  The previously defined actionPerformed( ) method is for listening the events on Button component, and itemStateChanged( ) method is for ComboBox component. It does the similar thing. When the ItemListener cathes an ItemEvent on the registered component, the itemStateChanged( ) method is called and the identifier of the trigger components passed as parameter. The number of gauges will be read from the ComboBox and when user make a change on ComboBox's value, this method will change the 'gauges_no' attribute.

- **public void paint (Graphics g)**

  When a graphic or a picture wanted to be drawn on the canvas, this method should be called. In some conditions, the system can also make a call to this method automatically. The average graphic will be drawn by this method.

- **public void report ( )**

  This method will be called when the Report button pressed and will show the report_frame.

4.1.2.3.1.2 General

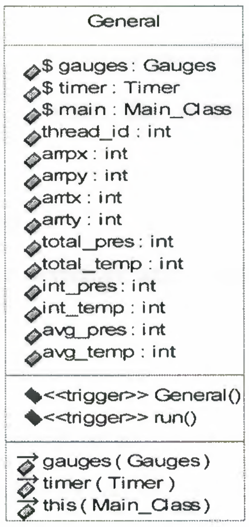


Figure 4.6. General class

Attributes :

- **int thread_id**

  Just after an instance of the General class created, an id will assigned to the thread and this attribute (thread_id) will keep this id.
- **int arrpx[], int arrpy[], int arrtx[], int arrty[]**

  For drawing the graphics, drawPolyline (int x[], int y[], int line_number) method will be used. This graphic method takes three arguments;
  - x[] : is the array that keeps the x coordinates of the lines.
  - y[] : is the array that keeps the y coordinates of the lines.
  - Line_number : number of lines that will be drawn.

  That is the reason why these arrays are defined. arrpx[] and arrpy[] will keep the x and y coordinates of the pressure graphic. arrtx[] and arrty[] will keep the x and y coordinates of the temperature graphic.
- **int int_pres, int int_temp**

  These will keep the values for temperature and pressure that are read from the Randomizing class.

- **int total_pres, int total_temp**

  Every read values for pressure will be summed to the total_pres and every read values for temperature will be summed to the total_temp. These will be used in finding the average values when the average period comes and after, will be resettled.

- **int avg_pres, int avg_temp**

  These will keep the average pressure and temperature.

Methods :

- **public General (Main_Class param_main)**

  This method is the constructor of the class. Here, the attributes will be initiated.

- **public void run( )**

  This is the method of Java's Thread class and should be overridden by the General class. When an instance of the thread class created, system first calls the constructor and then calls the run( ) method. For continues flow of execution, inside this method there should be an infinite loop. The operations should be places in this infinite loop.

4.1.2.3.1.3 Gauges

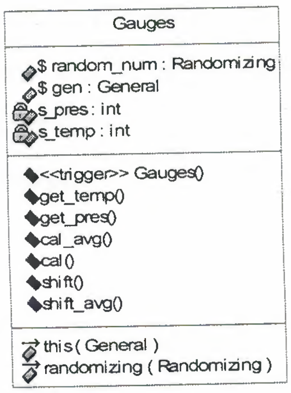


Figure 4.7. Gauges class

Attributes :

- **Private String s_temp, Private String s_pres**

  These attributes will keep the read values from the Randomizing class in string type.

Methods :

- **public Gauges (General p_gen)**

  This method is the constructor of the class and will be used only for initializing the attributes.
- **public int get_temp( ), public int get_pres( )**

  As the attributes s_temp and s_pres are private, they can be accessed from outside of this class. These methods first set these attributes by getting the random numbers from the Randomizing( ) class then returns the values of these attributes.
- **public void Cal_avg( )**

  The thread created for the average, will call this method to calculate the graphic coordinates.
- **public void cal( )**

  The threads created for the gauges objects, will call this method to calculate the graphic coordinates.
- **public void shift ( ), public void shift_avg ()**

  As the graphics will be too long to be displayed, after some time they should be shifted left to open space for the new coming lines. These methods will realize this for gauges and average graphics.

4.1.2.3.1.4 Randomizing

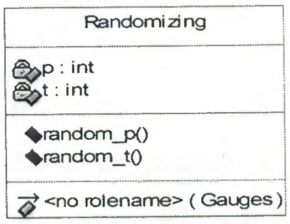


Figure 4.8. Randomizing class

Attributes :

- **private int p,   private int t**

  These are the private attributes that keep the temperature and pressure.

Methods :

- **public int random_p( )**

  This method generates a random number for pressure and returns this number.
- **public int random_t( )**

  This method generates a random number for temperature and returns this number.

4.1.2.3.1.5 Timer

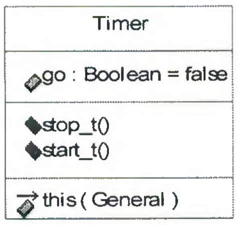


Figure 4.9. Timer class

Attributes :

- **boolean go=false**

  This attribute will be used to control the thread state. If the attribute is false, the thread will sleep and if the attribute is true, the thread will continue to flow.

Methods :

- **synchronized void stop_t( )**

  When this method is called over an instance of Timer class, the owner thread of the instance will sleep.
- **synchronized void start_t( )**

  When this method is called over an instance of Timer class, the owner thread of the instance will be notified and it will continue to flow.

4.1.2.3.1.6 Drawing

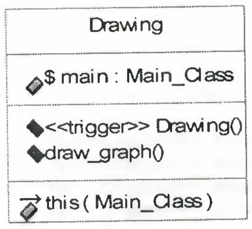


Figure 4.10. Drawing class

Methods :

- **Public Drawing(Main_Class main_p)**

  This is the constructor of the class and it only takes the parameter and assigns it to a local variable with the same type.

- **public void draw_graph (Graphics g, int thread_id)**

  This is the method where actual drawing of the average graphic is realized. While drawing, the 'g' attribute, which comes as an argument from the Main_Class (Frame), should be used. The thread_id attribute is the which_thread attribute in Main_Class that is keeping the id number of the thread that called the paint( ) method of Mian_Class. This attribute tells which threads coordinate arrays (arrpx[], arrpy[], arrtx[], arrty[]) will be drawn.

4.1.2.3.1.7 myexception

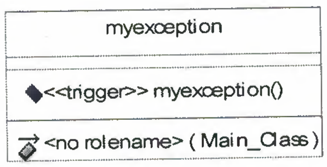


Figure 4.11. myexception class

Methods :

- **public myexception(String Message)**

  This method is the constructor and will just display the string variable message on the screen in dialog box.

4.1.2.3.1.8 Clock

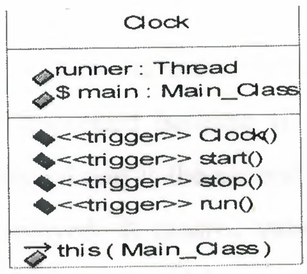


Figure 4.12. Clock class

Methods :

- **Public Clock(Main_Class main)**

  This is the constructor will just assign the main argument to a local variable.

- **Public void start( )**

  This class will implement the runnable interface and must override the methods of the interface. Start( ) method is one of these methods. Thread is started with this method.

- **Public void stop()**

  This method is another one which must be overridden. Thread is stopped with this method.

- **Public void run( )**

  This method also must be overridden. In an infinite loop, the paint( ) method of the Main_Class is called to display the clock in every average period interval.

4.1.2.3.1.9 Report

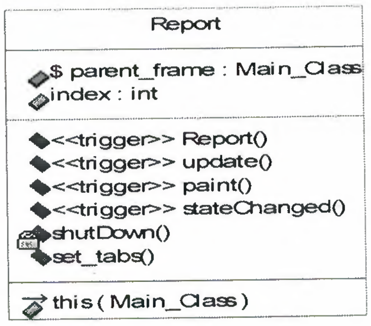


Figure 4.13. Report class

Attributes :

- **int index**

  This attribute will be used to keep the index number of the selected tab on the screen. The selected tab is important because if the first tab selected the first thread's graphic will be displayed and if the second tab is selected then the second thread's graphic will be displayed. It means, value of index will tell us which element of the General [1..6] array should be accessed to get the coordinates and draw the graphic.

Methods :

- **public Report( )**

  This is the constructor and the global attributes will be initialized in here.
- **public void paint(Graphics g)**

  This is the method where the drawing of the graphics will be done.
- **public void stateChanged(ChangeEvent e)**

  The events on the TabbedPane should be caught. So the class should implement the ChangeListener interface and override the stateChanged( ) method. When a ChangeEvent occur on one of the registered TabbedPane, this method will be called. The index attribute will be set with the selected index of tabbedPane and paint() method will be called to draw the entire graphic.

4.1.2.4 Structuring Sequence Diagrams

The Collaboration and Class diagrams gives us a detailed idea about the class structures and the works done in each class but they does not tell anything about how the work done. The Sequence diagrams will clear this fog and let programmer to do deeper design of the system.

4.1.2.4.1 Sequence Diagram-1 : Main_1

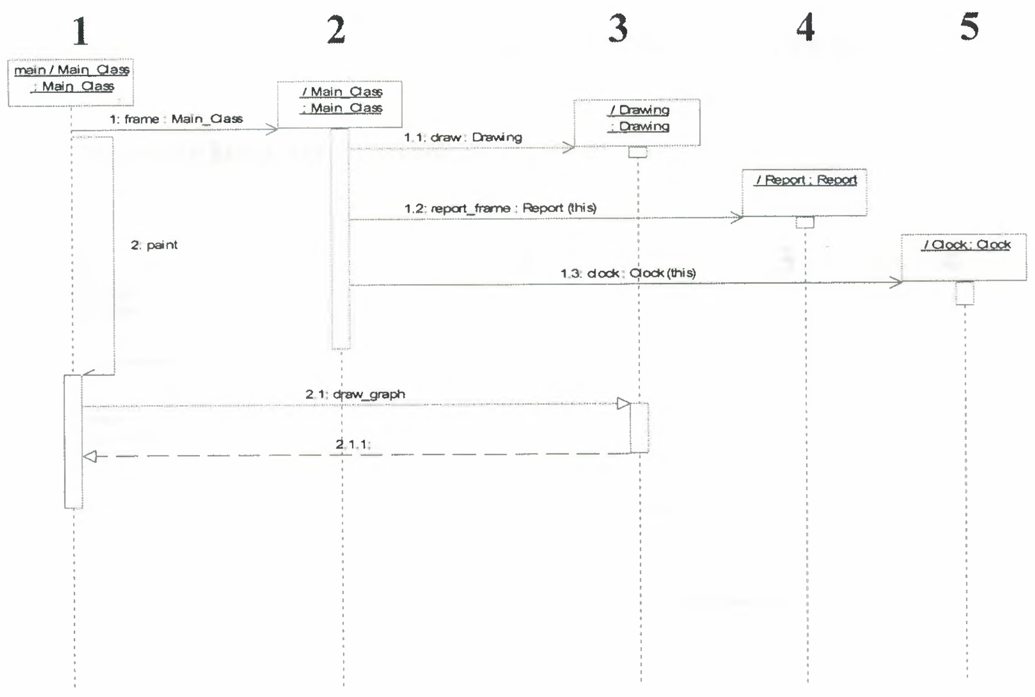


Figure 4.14. Main_1 Sequence diagram

This diagram is a general model of Main_Class and shows generally what the Main_Class should do in which order.

The instance-1 is the main( ) method in Main_Class and it is called by the system and can be assumed as the System itself. It can be seen that the first thing it does is making a call to the constructor of the Main_Class by creating an instance of the class with the name 'frame'. As it is assumed that this method is the system itself, it also can make a call to the paint( ) method when it is necessary.

The instance-2 is the constructor of the Main_Class as it is said. Generally the main job of constructors is to initialize the attributes. In addition to this, it can be seen that it builds the associations by initializing the instances of necessary classes; Drawing, Report and Clock.

After the constructor establishes the associations, the Main_Class can access any of the public attributes and methods in any of these classes. This means the paint( ) method in Main_Class can make a call to the draw_graph( ) method of Drawing class.

4.1.2.4.2 Sequence Diagram-2 : Main_2

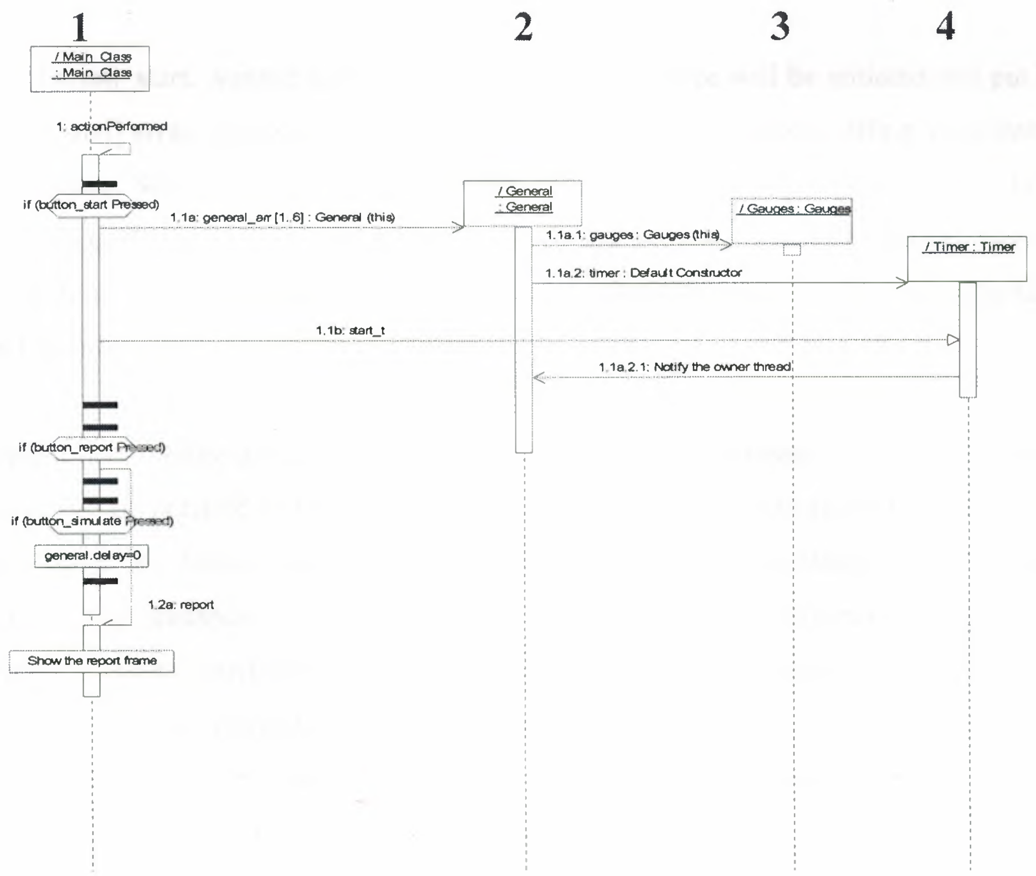

1
2
3
4
/ Main_Class : Main_Class
1: actionPerformed
if (button_start Pressed)
1.1a: general_arr [1..6] : General (this)
/ General : General
/ Gauges : Gauges
1.1a.1: gauges : Gauges (this)
/ Timer : Timer
1.1a.2: timer : Default Constructor
1.1b: start_t
1.1a.2.1: Notify the owner thread
if (button_report Pressed)
if (button_simulate Pressed)
general.delay=0
1.2a: report
Show the report frame

Figure 4.15. Main_2 Sequence diagram

As it is going to be complicated to represent and to understand the whole operations in a single Sequence diagram, it is better to divide the diagram to peaces. This is the second peace of the Main_Class sequence diagram.

After the constructor finishes its job, the system will begin to listen the registered components to catch the events on them. When any button clicked, system will catch it and make a call to the overridden actionPerformed method with sending the source components as a parameter. This parameter will be received and assigned to the variable 'e' by the method. Then, by checking the 'e' it will be decided what should be done. This checking process can be seen in instance-1.

If e = button_start, wanted number of General class instance will be initiated and put in the general_arr[] array. Instantiating an instance of a class simply means calling the constructor of that class. So in instance-2, it can be seen that the constructor of General class initiates an instance of Gauges and Timer classes. As call to the constructor of General class made several time, each instance of General class creates its own instances referred to the Gauges and Timer classes.

As mentioned before all created threads (instance of General classes) are initially sleeping and should be notified to flow. For this, a call should be made to the start_t( ) method of class Timer. As Main_Class has no association between Timer class, the call should be made over an instance of General class. So the general_arr[] array must be used for this purpose. General_arr[0].timer.start_t( ) command will notify the first thread. This process can be thought as firing the engine. You just press the starter and pistons are working continuously. Like this case, just one of the threads is triggered and all things will be done continuously inside them.

Return to the actionPerformed method. If e = button_report, report method will be called report ( ) method will show the Report frame.

If e = button_simulate, the sleep time of the threads will be zero millisecond instead of 1000 milliseconds and the experiment will over soon.

4.1.2.4.3 Sequence Diagram-3 : General_1

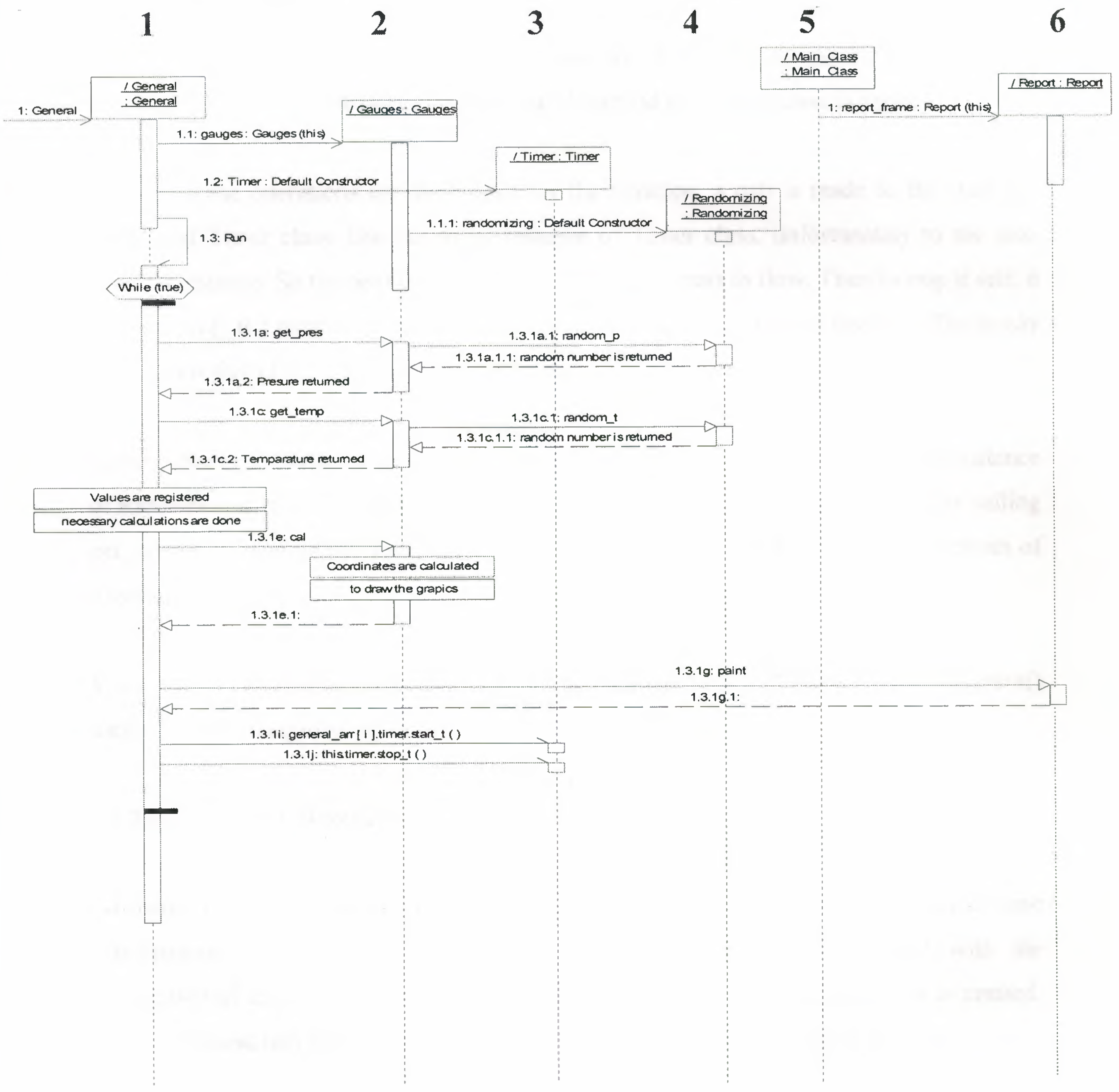

1
2
3
4
5
6
/ Main_Class : Main_Class
/ General : General
/ Report : Report
1: General
/ Gauges : Gauges
1: report_frame : Report (this)
1.1: gauges : Gauges (this)
/ Timer : Timer
1.2: Timer : Default Constructor
/ Randomizing : Randomizing
1.1.1: randomizing : Default Constructor
1.3: Run
While (true)
1.3.1a: get_pres
1.3.1a.1: random_p
1.3.1a.1.1: random number is returned
1.3.1a.2: Presure returned
1.3.1c: get_temp
1.3.1c.1: random_t
1.3.1c.1.1: random number is returned
1.3.1c.2: Temparature returned
Values are registered
necessary calculations are done
1.3.1e: cal
Coordinates are calculated
to draw the grapics
1.3.1e.1:
1.3.1g: paint
1.3.1g.1:
1.3.1i: general_arr [i].timer.start_t ( )
1.3.1j: this.timer.stop_t ( )

Figure 4.16. General_1 Sequence diagram

This diagram shows the detail of General class and its associations. In instance-1 the constructor of the General class creates an instance of Gauges and Timer class. Then the system automatically make a call to the run ( ) method of thread. To the end of experiment, for each second, the get_pres( ) method of Gauges class is called and this method returns a value for pressure. Then in a same manner the get_temp( ) method called and the temperature is returned. The returned values are displayed on the main screen and necessary calculations are done. Then the cal( ) method of Gauges class is called.

After all these operations are done once for the iteration, a call is made to the start_t( ) method of Timer class. But not to its instance of Timer class, unfortunately to the next thread's instance. So the next thread will be notified and start to flow. Then to stop it self, it calls the stop_t( ) method of the Timer class but this time over its own instance. The newly started new thread will also do the same process.

In the instance-2, it can be seen that the constructor of the Gauges class creates an instance of Randomizing class. When the General class requests a number from this class by calling get_pres( ) and get_temp( ) methods, it calls the random_p( ) and random_t( ) methods of Randomizing class.

When there is an internal or external call to the paint( ) method of Report class, the paint() method gets the required data form General class.

4.1.2.4.4 Sequence Diagram-4 : Threads

As there are multiple threads in the system, it is hard to represent and understand these synchronous threads in the previous diagrams. This diagram is prepared with the assumption of created two gauges. But as seen three instance of General class is created. This is because, two threads are for the selected gauges and one for taking average.

Each created instance of General class also creates an instance of Gauges class. The reason of this actually it is supposed that each gauges connected to different port address and this makes each Gauges instance has its own Randomizing instance.

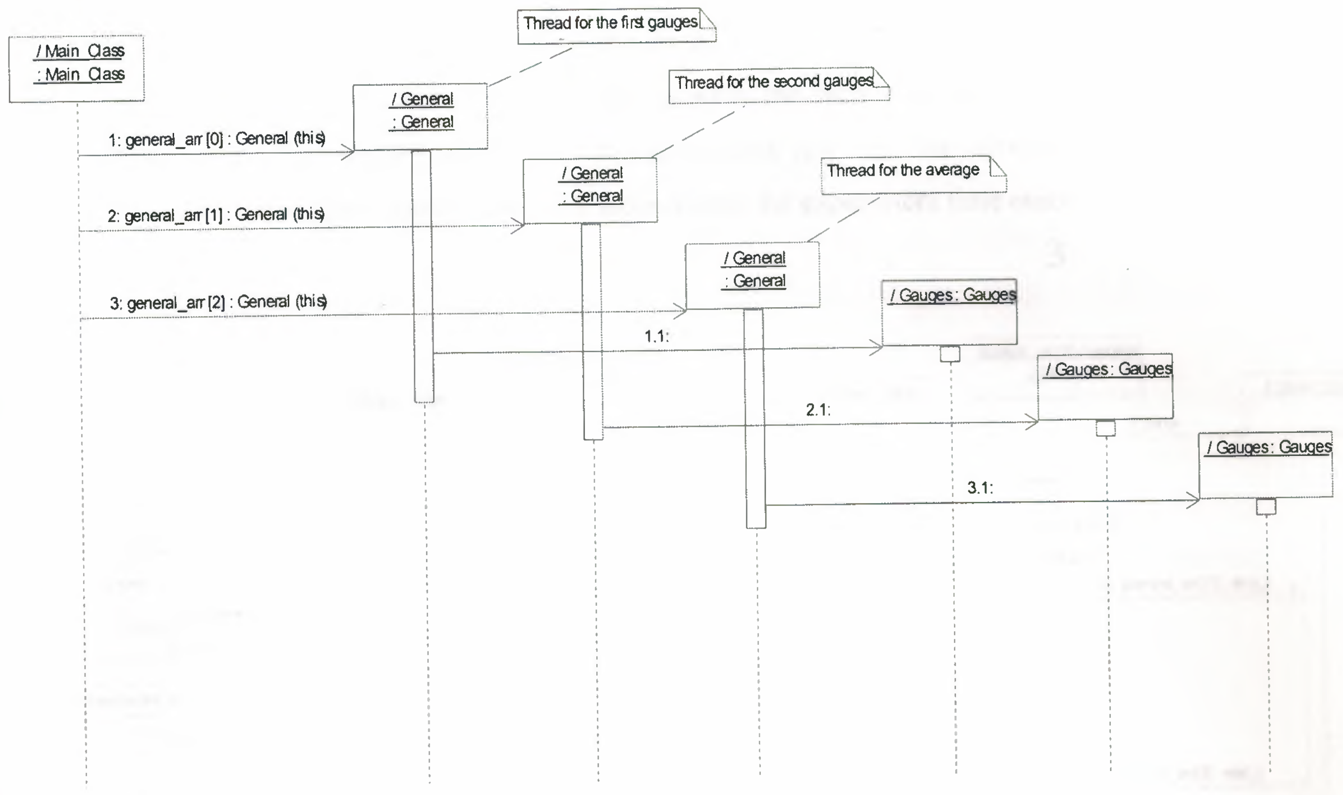


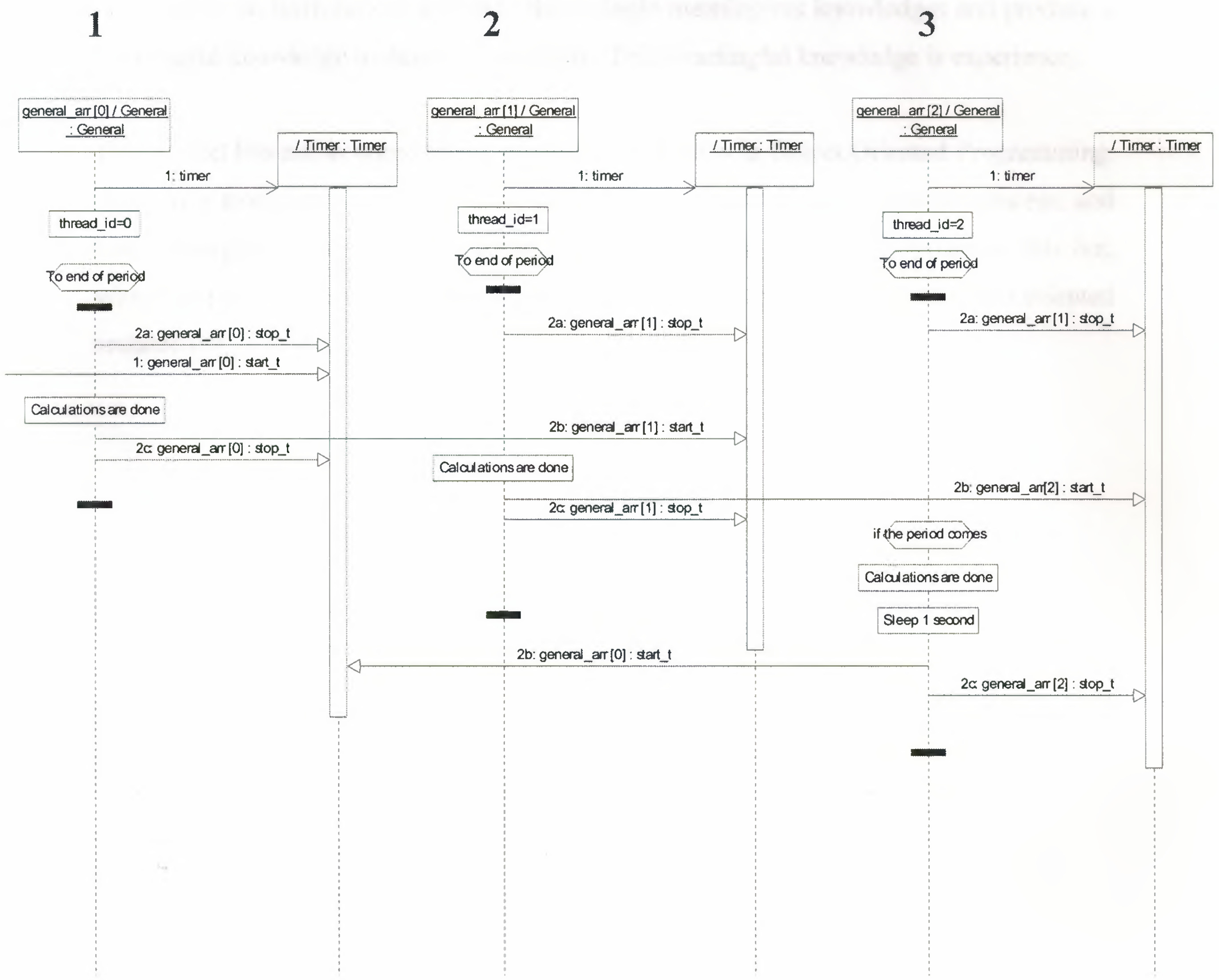
Figure 4.17. Treads Sequence diagram

5.1.2.4.5 Sequence Diagram-5 : Flow_of_Threads

In this sequence diagram each created General object is handled separately and their functions are examined detaily. Especially it is focused on their synchronization.

It can be seen that each instance of General class (thread) creates its own instance of Timer class. An id assigned to each thread with the variable name thread_id. After, each thread calls the stop_t( ) method of Timer class over its own Timer instance. This makes all thread to sleep until they are notified. Then from outside (from Main_Class) a call is made to the start_t( ) method of the first element in general_arr[] array. This notifies the first thread in the array and let if flow. After the first thread finishes its calculations, it notify the second thread by making a call to the start_t( ) method of the second element in the general_arr[]

array. Just after, it makes a call to the stop_t( ) method of itself and get into sleep until last thread notify it. This can be seen on the diagram. The instance-3 notifies the first thread before sleep itself. Be aware that last thread waits the second to pass before notifying the first thread. This waiting makes each thread to work just once for iteration in one second. This process will be repeated for each second until the experiment time ends.



1
2
3
general_arr [0] / General : General
/ Timer : Timer
1: timer
thread_id=0
To end of period
2a: general_arr [0] : stop_t
1: general_arr [0] : start_t
Calculations are done
2c: general_arr [0] : stop_t
general_arr [1] / General : General
/ Timer : Timer
1: timer
thread_id=1
To end of period
2a: general_arr [1] : stop_t
2b: general_arr [1] : start_t
Calculations are done
2c: general_arr [1] : stop_t
2b: general_arr [0] : start_t
general_arr [2] / General : General
/ Timer : Timer
1: timer
thread_id=2
To end of period
2a: general_arr [1] : stop_t
2b: general_arr[2] : start_t
if the period comes
Calculations are done
Sleep 1 second
2c: general_arr [2] : stop_t

Figure 4.18. Flow_of_Threads Sequence diagram

After all these deep analyzes and diagrams, constructing the code will be much easier.

CONCLUSION

First of all, the main objective of this project is to put forward the academic knowledge gained during undergraduate education period. We have learned lots of things here in the university but they were all theoretical and meaningless by their own. With this project, we have chance to learn how to combine these single meaningless knowledges and produce a meaningful knowledge to develop a software. This meaningful knowledge is experience.

This project lets me to improve my experience and skills in Object Oriented Programming. There is a more important thing than writing code in software development process, and that is designing. Writing code can be learned from books and anybody can do this but, without a powerful design it is nearly impossible to develop the software in object oriented architecture.

REFERENCES

[1] Java™ Developer's Reference *(Imprint: SAMS.Net)* (e-book)

[2] Java By Example By Clayton Walnum (e-book)

[3] JAVA Developer's Guide By Jamie Jaworski (e-book)

[4] Tricks of the Java Programming Gurus By Glenn L. Vanderburg (e-book)

[5] Java 1.1 Unleashed By Michael Morrison (e-book)

[6] Java AWT Reference By John Zukowski (e-book)

[7] Object-Oriented Software Development with UML By Zhiming Liu (e-book)

[8] UML Distilled By Martin Fowler (e-book)

[9] Mastering UML with Rational Rose 2002 By Wendy Boggs & Michael Boggs (e-book)