Chapter 5 Implementation

5.0 Implementation Model

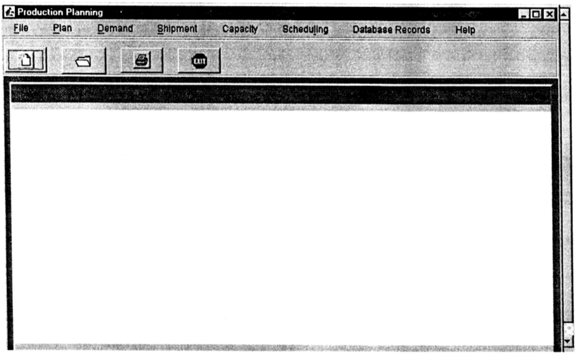
The implementation model consists of the source code, classes and the necessary documents based on the design of the system. An object-oriented language, **Java 1.2.1**, is used to write the source code of the customisable planning tool. Java is chosen as it supports encapsulation, classes, message passing as well as inheritance. It also supports superclass method overriding and explicit access to superclass methods. Java uses the keyword ***extends*** for class inheritance and also a ***package*** construct with an ***import*** statement, which allows the grouping of related classes that have convenient access to each other. It also provides an ***interface*** construct, that enables classes and their descendents, to define and ***implement*** several interfaces as a set of methods. An interface can then be used by other classes as a form of contract. Besides, the declarations, ***public***, ***private*** and ***protected*** provide visibility and access control between classes and packages.
Since Java is a fully object-oriented language, it makes the transition from design to code much more direct.

5.1 Application and Component Systems

The implementation process continues from the top-down design of the application systems that are traced to the component systems, which in short is implementing the architecture as a layered system. These are represented by the business entities which form the objects found in the main menu interface. Figure 5.1 shows the main interface of the implemented application systems. Appendix B shows in more detail the objects and classes which are part of the component systems that are hidden from the main screen. The design model that is traced to the use case model which consists of plan production, check capacity and schedule production use cases are integrated into the application subsystems. Each of the use cases is implemented with the priority to deliver the most important functionality first, for instance, to check for capacity before a production plan can proceed. Each increment is thus developed as one iteration through the software engineering process. On completion of an increment, the next set of use case is then refined and implemented.

All the entry screen classes for example **Customer User Interface, Product User Interface, Plan User Interface**, etc. have been derived from the swing class ***JDialog***. The main screen contains a main panel in the center. Interfaces which contain ***JTable*** are implemented as panels (derived from ***javax.swing.JPanel***) with ***JTables*** in it and form part of the data retrieval interface. In response to the menu events in the main menu, the corresponding panels will be added to the main panel. All the user interface classes used the ***DataManager*** class to access the database. For each action, the access is allowed according to the access restrictions set for the actor.

Figure 5.1 Main Screen of the Customisable Production Planning Tool



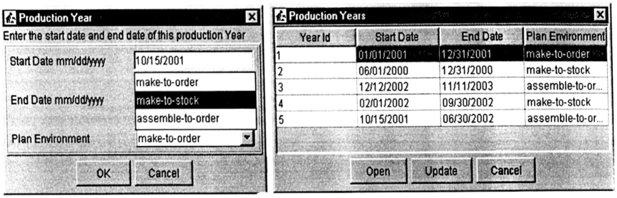
5.2 Customisation

The customisation part of the software as mentioned previously were implemented as variability points. The features of customisation include the three types of production environments which are Make-To-Order (MTO), Make-To-Stock (MTS) and Assemble-To-Order (ATO). Apart from these, users can also customise the calculation of their own inventory and forecast methods respectively, then the values will be passed to the respective inventory and forecast columns. To further secure the system, different levels of access can also be set for different users.

5.1.1 Production Environment

The three types of production environments which are Make-To-Order (MTO), Make-To-Stock (MTS) and Assemble-To-Order (ATO) are areas of customisation that allow the user to choose from. The way this is done is that in the ***ProdYear*** table there is a field called ***Prod_Env***. This field stores the environment name like MTO, MTS and ATO. The formulae to calculate the *Plan* is hard coded in the program as there are only three modes of environments. Appendix C shows the help file that contain the explanation for the formula used for the respective production environment. The user can select from the three environments when he starts a production year. He can also change the environment from one to another at any time. Then the environment name will be stored in the database table ***ProdYear*** against the ***ProdYear_id***. This name will be fetched by the program when calculating the plan function and it will calculate according to the corresponding formula for each of the production mode. Below shows an extracted part of the code for the implementation of the production environment. Whereas Figure 5.2 shows the customised production environment with their respective production years which has been implemented.

Figure 5.2 Customisable Production Environment


Production Year

Enter the start date and end date of this production Year

Start Date mm/dd/yyyy	10/15/2001
	make-to-order
End Date mm/dd/yyyy	make-to-stock
	assemble-to-order
Plan Environment	make-to-order

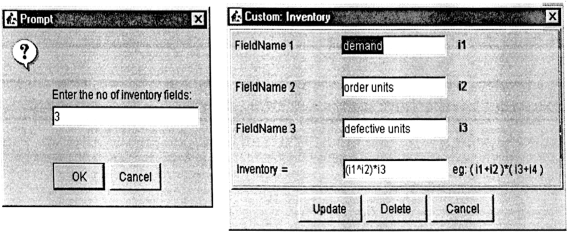
Production Years

	Year Id	Start Date	End Date	Plan Environment
1		01/01/2001	12/31/2001	make-to-order
2		06/01/2000	12/31/2000	make-to-stock
3		12/12/2002	11/11/2003	assemble-to-or...
4		02/01/2002	09/30/2002	make-to-stock
5		10/15/2001	06/30/2002	assemble-to-or...

5.2.2 Inventory and Forecast

Inventory and forecast fields and the formulae to calculate them are entered by the administrator or manager. Each time when the user starts a production year, he has to give the forecast and inventory details. These details will remain the same throughout that production year. The field names and the formulae are tokenised (breaking statements into individual pieces) using coma (,) and stored in a file in the format **<ProdYearId>.Inventory** and **<ProdYearId>.Forecast**. The user is supposed to fill in the fields which he has defined in the file, when he adds a *Plan* record. The program will prompt the user to enter the field values for the field names which he had entered at the time of a new production year. The program will calculate the inventory and forecast according to the formulae given and store it in the corresponding fields in the ***Plan Detail*** class. The actual values of the custom fields will be stored in a ***CustomTab*** class for future modification. The calculation is being done using a third party package which is known as ***eval.jar*** , a java archive compressed file, meant for computation purpose. The ***evaluate()*** function takes an arithmetic expression for example: (41/2)*(3+49) as the parameter and returns the result. The result will automatically be passed to the inventory and forecast columns inside the ***PlanDetail*** class. Whatever that is decided will be fixed throughout the production year. Figure 5.3 shows the implementation of the customisation of inventory. The user can key in any number of fields required for the calculation of the inventory and then formulate a function in arithmetic expression based on the number of fields given.

Figure 5.3 Customisation of Inventory

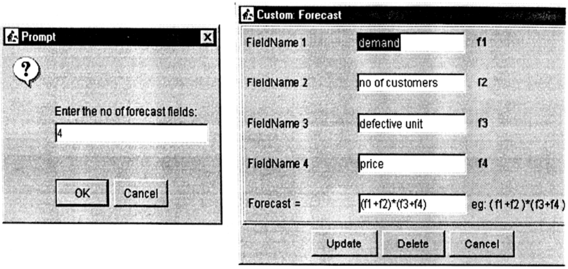


Enter the no of inventory fields: 3

FieldName 1: demand — i1
FieldName 2: order units — i2
FieldName 3: defective units — i3
Inventory = (i1^i2)*i3 — eg: (i1+i2)*(i3+i4)

The customisation process for forecast method is similar to the inventory method. Figure 5.4 shows the implementation of the customisation of forecasting method. The user can also key in any number of fields required for the calculation of forecast method and then formulate a function in arithmetic expression based on the number of fields given.

Figure 5.4 Customisation of Forecast



Enter the no of forecast fields: 4

FieldName 1	demand	f1
FieldName 2	no of customers	f2
FieldName 3	defective unit	f3
FieldName 4	price	f4
Forecast =	(f1+f2)*(f3+f4)	eg: (f1+f2)*(f3+f4)

In addition, ***jfreechart.jar***, which is another third party package is also used for displaying the graph for the year-to-date production plan. The graph displays the total production yield against the total production planned for the whole production period. Figure 5.5 shows the total *Year-To-Date* values of planned and actual production figures.

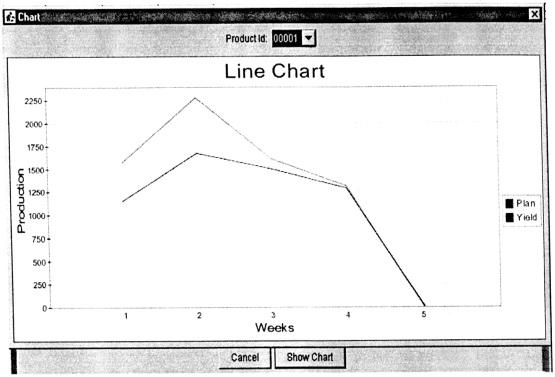


Figure 5.5 The Total Year-To-Date Production Values

5.2 Data Storage Implementation

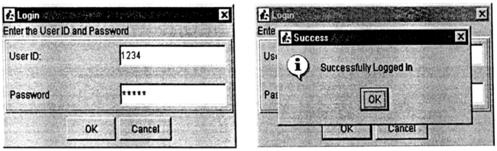
The storage management is implemented through two parts. The first being the implementation of the database design and the second part is the data retrieval interface. The database is a three-tier setup. The client interface acts as the first tier and the MS Access database acts as the third tier. Interaction with the database is done through the SQL. For connectivity to the database, we use a ***JdbcOdbcDriver***. A class ***DataManager*** implements the middle-tier features. It accesses the database and formats the results to display in the client side. The ***DataManger*** class fetches the results from the database and converts them into the form of ***java.util.Vector*** to display it in the client interface. ***Vector*** is a class which can store any number and any kind of objects in it. It is like an array which dynamically increases its size. When a select query is executed, the ***DataManager*** class takes all the results from the table and adds each row information in the form of a *vector*, so as to display it in the ***JTable***. As most of the interfaces in the client side consist of ***JTable*** , returning the results as *vector* is a good choice. The ***JTable*** has a constructor which takes ***Vector*** as the parameter in the place of tuples. The database connection through the ***DataManager*** class is to the datasource is as follows :-

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                    DriverManager.getConnection("jdbc:odbc:Production");
```

5.3.1 Login Access

Login access to the database is restricted through access privileges set by the administrator or manager. The manager must first login as an authorised user. Figure 5.6 shows the login window into the system. The manager can also grant and deny permissions to perform any action in the system.

Figure 5.6 Login Window



Enter the User ID and Password

User ID: 1234

Password *****

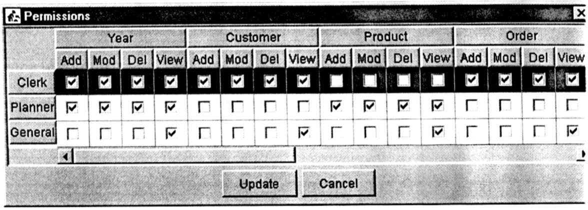
Successfully Logged In

A ***RegUserUI*** class is coded to show the different levels of users' access into the database. Below is a sample code of this class. Figure 5.7 shows the interface for RegUserUI class whereby only the manager can set the different levels of access for different users.

```
package produi;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.text.*;
import javax.swing.table.*;
import java.util.*;

public class PermissionUI extends JDialog implements ActionListener{

      public ProductMan parent;
      public JTable table;
      Vector vData;
      SimpleDateFormat dFormat;

      public PermissionUI(ProductMan parent){

            super(parent,"Permissions");
            this.parent = parent;
            getContentPane().setLayout(new BorderLayout());

            //////////////////////// DATA COLS
 Vector vCols = new Vector();

          for(int i=0;i<9;i++){

            vCols.addElement("Add");
```

```
  vCols.addElement("Mod");
  vCols.addElement("Del");
  vCols.addElement("View");

}
Vector vData = null;
try{
  vData = parent.dbMan.getAccessData();
}
catch(Exception e){

  e.printStackTrace();

 }

     ListModel lm = new AbstractListModel() {
     String headers[] = {"Clerk", "Planner", "General"};
     public int getSize() { return headers.length; }
     public Object getElementAt(int index) {
       return headers[index];
     }
   };
```

Figure 5.7 Permission for Use to Different Users



	Year				Customer				Product				Order			
	Add	Mod	Del	View	Add	Mod	Del	View	Add	Mod	Del	View	Add	Mod	Del	View
Clerk	✓	✓	✓	✓	✓	✓	✓	✓	☐	☐	☐	☐	✓	✓	✓	✓
Planner	✓	✓	✓	✓	☐	☐	☐	☐	✓	✓	✓	✓	☐	☐	☐	☐
General	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓	☐	☐	☐	✓

5.4 Test Procedures

The customisable production planning tool is tested with the intention of verification and validation. This is to ensure that this customisable software conforms to its specifications and have met all the functional specifications set out from the start as well as any added requirements during the development process. In the development of this customisable tool, the test model here is used to confirm the validity of the other models produced during software engineering. The testing process is done through each use case instances. The instance is an object which has behaviour and state and the outcome of this test execution is a test result. The test result will then validate the consistency of each model and its mapping to the other models. Testing also includes the subsystems through to the whole layered system and is carried out in all iterations from as early as the design phase as this helps refine and better understand the requirements. In this case testing is started with subsystem testing and then continued with integration testing, function testing and finally system testing which in short, refers to the bottom-up testing approach.

In object-oriented paradigm individual operations or subsystems are regarded as components. They are tested as part of the class and the class or an instance of a class (object) then represents the smallest testable unit or module. A class and its operations is the module most concentrated on in the object-oriented environment. From here it expands to other classes and sets of classes. Object-oriented test cases need to concentrate on the states of a class. To examine the different states, the cases have to follow the appropriate sequence of operations in the class. Class, as an encapsulation of attributes and procedures that can be inherited, thus becomes the main target of object-oriented testing. In this customisable planning tool, all the classes within the component systems were tested accordingly. The state of each class is tested according to the sequence of operations defined in the respective classes. The customisable features such as the variation points were given more emphasis as this involves perceiving clients' input actions and operations. The overall operations of a class were tested using the conventional white-box methods and techniques such as message passing, loop and data flow. White box method is most widely utilised in these subsystem testing to determine all possible paths within the application system so as to execute all loops and to test all logical expressions. Because of inheritance, testing individual operation separately or independently of the class would not be very effective, as they interact with each other by modifying the state of the object they are applied to [Binder, 1994]. Since there is interaction between objects, the tests were designed to test each new context and re-test the superclass as well to ensure proper working order of the objects.

As use cases are basically descriptions of how the system is to be used, therefore each of the use cases was tested. In this function test process that is black-box in nature, real user data were used to do the testing of the overall functionality of the customisable production planning system. This includes testing of all the interfaces and the interaction of the actors with the use cases. The specifications for this function test were very detailed as every aspect of the software system was being tested. The system's functions were evaluated as new requirements were added according to what needed to be improved. Since the tests were iterative from design to implementation, each phase is refined until all the customised functions worked according to their actual functionalities.

The final stage of the test process was integration and system testing for the whole layered system. This type of test involves examination of the whole customised system which includes the software components, all the hardware components and any interfaces. All the use cases were integrated and their interfaces interact so as to complete the workings of the entire system. The whole layered system is checked not only for validity but also for verifying that it has met all the objectives and functionality of this customisable production planning tool meant for the manufacturing sector.

A help file is also generated using MS Help Workshop, a software available from the internet, to display a comprehensive help file for users' reference. All the explanations are given about the workings of the production planning system that will help user understand the functionality of this software. Appendix D shows a print out of the help file.