





***DESIGN REPORT  
FOR  
THE PROJECT  
INVENTORY CONTROL SYSTEM FOR  
CALCULATION AND ORDERING OF  
AVAILABLE AND PROCESSED  
RESOURCES***

*(November 12, 2012)*

**GROUP 9**

 ***SIMANT PUROHIT***  
 ***AKSHAY THIRKATEH***  
 ***BARTLOMIEJ MICZEK***  
 ***ROBERT FAIGAO***

## INDEX

<b>1. Introduction</b>	
<b>1.1 Purpose of the system</b> .....	<b>4</b>
<b>1.2 Design goals</b> .....	<b>4</b>
<b>1.3 Definitions, acronyms, and abbreviations</b> .....	<b>6</b>
<b>1.4 References</b> .....	<b>6</b>
<b>2. Current software architecture</b> .....	<b>7</b>
<b>3. Proposed software architecture</b>	
<b>3.1 Overview</b> .....	<b>7</b>
<b>3.2 Subsystem decomposition</b> .....	<b>8</b>
<b>3.3 Hardware/software mapping</b> .....	<b>10</b>
<b>3.4 Persistent data management</b> .....	<b>11</b>
<b>3.5 Access control and security</b> .....	<b>12</b>
<b>3.6 Global software control</b> .....	<b>13</b>
<b>3.7 Boundary conditions</b> .....	<b>14</b>
<b>4. Subsystem services</b> .....	<b>16</b>
<b>5. Database System</b> .....	<b>17</b>
<b>6. Object design trade-offs</b> .....	<b>18</b>
<b>7. Interface documentation guidelines</b> .....	<b>19</b>
<b>8. Packages</b> .....	<b>21</b>
<b>9. Class interfaces</b> .....	<b>23</b>
<b>Bibliography</b> .....	<b>34</b>
<b>Glossary</b> .....	<b>35</b>

## List of Figures

Figure 1: Subsystem Decomposition .....	8
Figure 2: Hardware/Software Mapping .....	10
Figure 3: Services Diagram.....	15
Figure 4: Database Schema.....	16
Figure 5: Access Specifiers.....	18
Figure 6: Packages Diagram.....	20
Figure 7: Class Diagram (Overview) .....	22
Figure 8: Ingredient Class Diagram.....	23
Figure 9: AddIngredient Class Diagram .....	24
Figure 10: Recipe Class Diagram .....	25
Figure 11: Vendor Class Diagram.....	26
Figure 12: Prediction Class Diagram.....	27
Figure 13: AddRecipe Class Diagram.....	28
Figure 14: RemoveRecipe Class Diagram .....	29
Figure 15: UpdateRecipe Class Diagram.....	30
Figure 16: Updates Class Diagram.....	31
Figure 17: Occasion Class Diagram .....	32
Figure 18: Orders Class Diagram .....	33

## List of Tables

Table 1: Subsystem Description .....	9
Table 2: Access Matrix .....	12
Table 3: Boundary Exception Cases .....	14

# 1. Introduction

## 1.1 Purpose of the System

A case study at “Guckenheimer” (an on-site corporate restaurant management and catering company) cited issues regarding a basic resources requirement list that has to be maintained manually by the staff. To keep track of their inventory levels they have to calculate a list of the groceries utilized during a course of time, calculate and analyze the requirements for the future, and place their next order to the vendors if needed. This process takes up a lot of time and human effort, and is also prone to human error.

This poses a problem of a situation that the staff at “Guckenheimer” as well as many other restaurants faces. It takes up a lot of time to manually keep track of sales and place correct orders to vendors, wasting useful labor in trivial works. A product which would assist in tackling the above mentioned problems would prove to be fruitful to clients such as “Guckenheimer” and similar enterprises as this product would help convert the unproductive time to something more useful, by removing the unnecessary error prone complications and efforts.

## 1.2 Design Goals

- **Low Response Time:** The main functionality of the system involves updating and reading the data from the database for different entities such as ingredients, recipes vendor etc. Thus the time required to retrieve/ update/ add data to the database should be minimum and preferably should be in the range of 2-5 seconds or lesser.
- **High Robustness:** The system should constantly check the user input at all instances that could generate errors in the program. For instance-
  - The system should be able to check input values for the amount of ingredients required for the recipe and should make sure the user enters a numeric value in the input box and the system shows an error and asks the user to re-input if in a perfectly validated field an improper data type is inputted.
  - The System should have validated input data fields and must put a constraint on the inputted names of recipe, ingredient, vendor, occasion etc. to ensure no duplicate entries are added in the database. This ensures the robustness of the maintained database.

- The system should verify all the inputs by the user by using a confirmation dialog box before processing and making changes to the data.
- **High Reliability:** The reliability of the system depends upon its ability to replicate the specified behavior. The safekeeping of the data is essential so as a result a backup of the levels is generated and stored in the warehouse. There are numerous factors on which reliability can be defined as for example, the specifications mention that the updating of database or the notification of a successful update must be carried out within 2-5 seconds of initiation and the system must adhere to these specifications to be called a reliable system. Similarly, the system should be able to achieve performance in lieu with the specifications mentioned.
- **Low fault tolerance:** The system works on sensitive data and therefore any fault in the functioning of the system will hinder accurate updating or reading of data. This could lead to invalid entries in the database. Thus, the system should have low fault tolerance. This is in tandem with the design goal of high robustness as the validation checks to ensure correct inputs from the user implies that the fault tolerance of the system is low.
- **Security:** The system must provide a login functionality to the *Manager* as the manager is the authenticated controller of the system and any other user is not permitted to use the system functionality and make changes in the database. Thus proper user authentication should be necessary before system launch.
- **High Extensibility:** The design of the system should be such that any future improvement can be added with no or minimum improvements. It is in one's best interest to always give space for future enhancements. For instance, right now there is no class that will help the user to manipulate prediction of values and the current system only predicts the ingredient usage for a certain date, but a feature can easily be added to incorporate prediction of recipe usage, order prediction etc.
- **Low Adaptability:** The system is designed to work on the domain of inventory control and management in the restaurant and catering industry. The functioning of this project is limited only to these particular businesses which have similar functioning and thus it would then be subject to structural re-modification in order to to apply it in some other application domain.
- **High Readability:** The system code should be properly commented so as to explain the functionality of the code fragments. The code comment should explain the function or

task the code fragment performs and the result and the return value of the corresponding function or task should also be mentioned.

- **High Traceability:** The coding scheme of the system should be such that it could be traced back to its requirements specifications. This will enable high traceability of the code of the system.

### **1.3 Definitions, acronyms, and abbreviations**

- **Manager:** The manager implies the manager of the restaurant/company who handles all the administrative works.
- **Recipe:** This is the menu item that the restaurant/company provides to its customers.
- **Ingredient:** This is the entity that the recipe is composed of.
- **Vendor:** This is the company that provides the restaurant/company with the required ingredients.
- **Order:** Order is the list of ingredients and the quantities that is or is to be requested from the vendor.

### **1.4 References**

- Project proposal document: Submitted on 21<sup>st</sup> September 2012.
- Software Requirements Specification Document: Submitted on 19<sup>th</sup> October 2012.

## **2. Current Software Architecture**

“Guckenheimer” (an on-site corporate restaurant management and catering company) follows a system where the basic resources list needs to be manually calculated at the end of a certain time period by the staff. They must accordingly check the inventory levels for determining if they are below the threshold level then orders are processed to the vendors. This sort of system not only leaves a lot of room for human error, but is also incredibly time consuming. The lack of a centralized database also creates an issue when it comes to keeping track of inventory levels as well as past trends in ingredient requirements. The system also relies on human intuition and guesswork to place the correct orders for the following week, which will not be as precise as an algorithm designed for this purpose.

## **3. Proposed System Architecture**

### **3.1 Overview**

We propose to develop software that keeps track of inventory in the “back of house”, or kitchen, and updates it according to daily sales. Each food item is linked to respective resources (or ingredients) and as each product is sold the ingredients utilized in making that product are also utilized. These changes in inventory are kept track of through utilizing a database.

We propose to keep track of each and every ingredient by dynamically linking it to the product and as a result create a dependent relationship to that product. At a specific time period (typically the end of the week); if the inventory is below the threshold level, order forms to the specific vendors are generated in order to restock the required items for the next week. The project also makes smart predictions on required inventory for the following week based upon the predicted climate and possible occasions or events that may influence near future sales. At the end of the week, the software takes into account all threshold levels, predictions, and other factors to generate an order form, which after being verified by the manager is sent out to the vendors.

### 3.2 Subsystem Decomposition

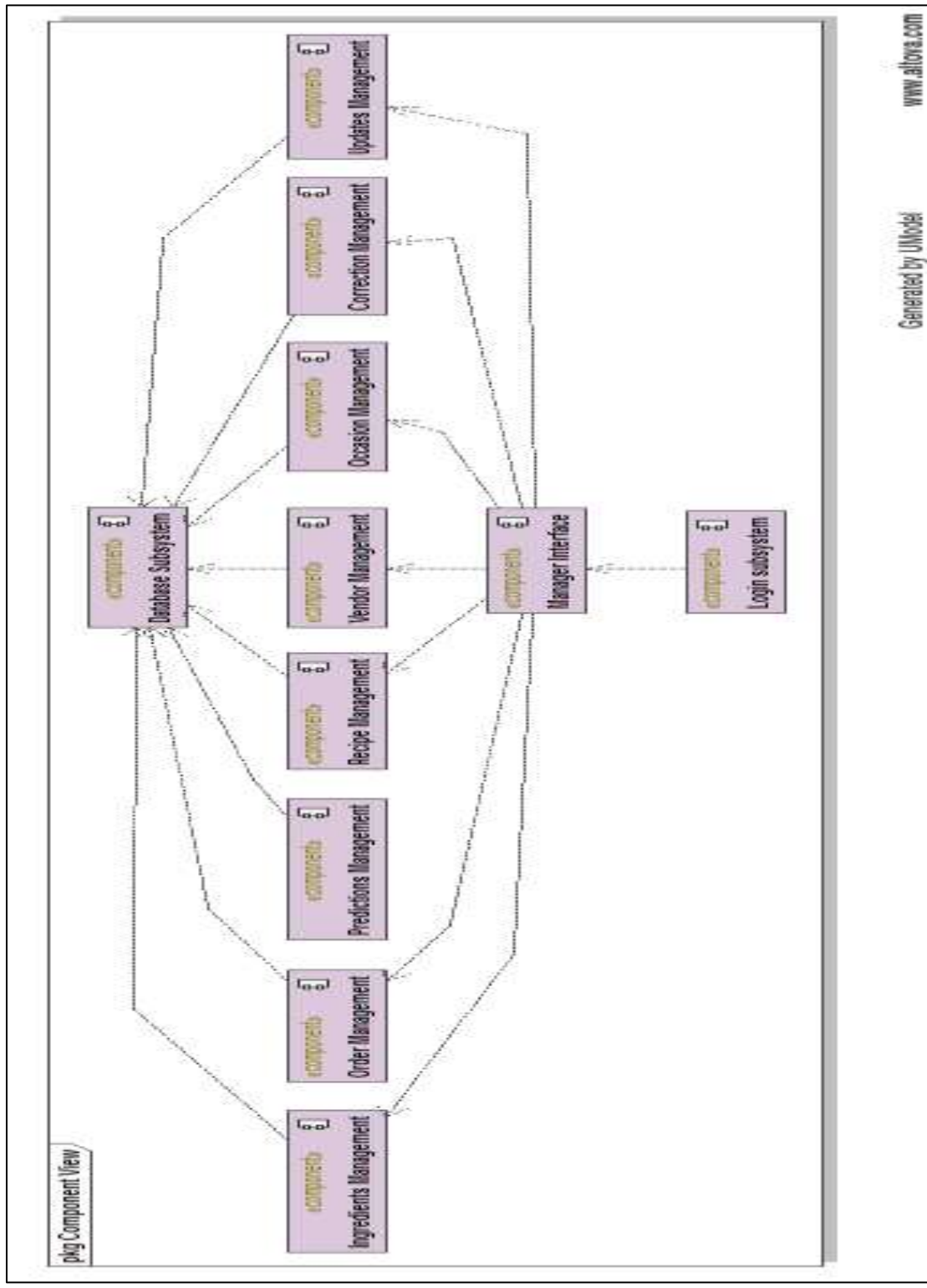


Figure 1



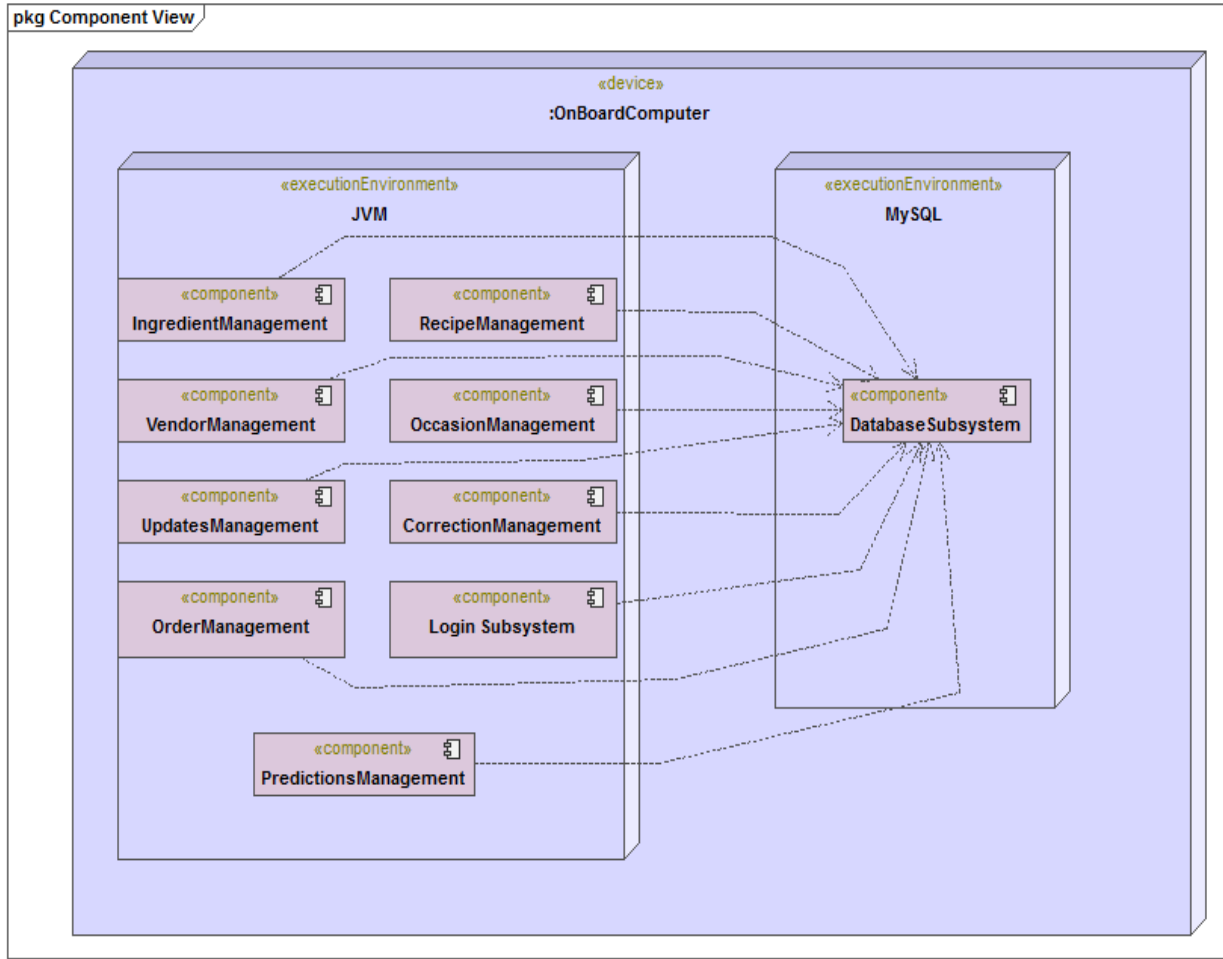
## Subsystem Description

ManagerInterface	This subsystem defines an interface between the user and the system. The user through this interface can access and execute different functions on the various subsystems.
IngredientsManagement	This subsystem provides services to manage the ingredients inventory of the system. This subsystem provides services such as providing list of available ingredients, providing details of individual ingredients in terms of the current inventory levels, threshold levels etc. This subsystem requires the services of Database subsystem to retrieve required details.
RecipeManagement	This subsystem provides services to manage the Recipes in the inventory. This subsystem provides services such as adding/updating/removing a recipe to/from the inventory. This subsystem requires the services of Database subsystem to retrieve the list of ingredients that make up the recipe.
VendorManagement	This subsystem provides services to manage the vendor that deliver Ingredients. This provides services such as providing details of the vendor, providing list of ingredients that the vendor supplies. This subsystem communicates with the Database subsystem to retrieve the list of vendors and the ingredient list to match them with the vendor names.
PredictionManagement	This subsystem provides prediction of usage per ingredient to the user. This subsystem communicates with the Database subsystem to access past data of used resources from the database and then apply prediction algorithm on the retrieved data to give estimates of usage to the user.
OrderManagement	This subsystem provides services to generate orders for vendors for the ingredients that are below threshold levels. This subsystem also provides like editing a generated order and cancelling a generated order.
OccasionManagement	This subsystem provides services to the user to add an occasion date to the system so that the system prepares itself for an upcoming event on which day the sales will be more than usual day sales. This subsystem requires the services of Database subsystem to update/remove occasion days from the database.
CorrectionManagement	This subsystem provides services to the user to correct the levels of inventory and avoid inventory slips. The user here uses this service to match the actual inventory levels with the inventory levels in the system.
UpdatesManagement	This subsystem provides services to the user to perform updates on the database.
DatabaseSubsystem	This subsystem connects to the database and provides requested database to the other subsystems that request data from it.

Table 1

### 3.3 Hardware Software Mapping

The system runs on a standalone system without the need of any external server connection or internet connection. Thus the hardware requirement of the product is a personal computer which meets the requirements mentioned in the specifications. The product is programmed in Java programming language and uses MySQL for database service. Thus the client computer requires the installation of JDK 1.6 and MySQL server on its machine. The mapping between the hardware and the software can be interpreted by the following diagram.



Generated by UModel

www.altova.com

Figure 2

## **3.4 Persistent Data Management**

### **Persistent Objects**

The main data entity that is persistent in the system is objects of class Ingredients. The objects of class Ingredients are used by various classes to function. For example, the class Recipe uses the objects of Ingredients class to define the contents of recipe with the name of ingredients that are accessed by the objects of the class of Ingredients.

The object of the class Recipe have also to be classified as persistent objects even though the class derives some of its properties from the Ingredients class. The reason for this is the classes AddRecipe, RemoveRecipe, UpdateRecipe and Occasion are derived from the Recipe class and require the object of the recipe class for their functionality.

Similarly, the objects of class Vendor have to be persistent as it also forms a building block of the whole database system. The Vendor class objects give the list of vendor along with the ingredients they provide. Thus the Vendor class uses the objects of the Ingredients class for its functionality.

In short, the objects of classes Ingredients, Recipe and Vendor are connected and depend highly upon each other for their functionality. Additionally, other classes defined in the class diagram of the system depend upon the data provided by the above mentioned classes for their functionality.

### **Storage Strategy**

The database is created and maintained in the open source environment MySQL. The subsystems connect to the database via JDBC API. Using an open source database management system enables us to reduce cost of development of the system plus it allows us to maintain the database on the same machine on which the other subsystems are functioning. The only thing required is a JDBC connector driver to setup and maintain connection to the database. The type of database used is Relational database, as using a flat file database would prove tedious if used on such highly connected data entities that we use in our system.

### 3.5 Access Control and Security

#### Access Matrix

<b>Orders</b>	<b>Actors →</b>	<i>Manager</i>
Recipe		addRecipe() removeRecipe() updateRecipe()
Ingredients		getIngredientsList() addIngredient() updateIngredient()
Vendor		getVendorDetails() getIngredientListFromVendor()
Prediction		getPredictedUsage()
Updates		updateAfterSales() updateAfterReceiving()
Occasion		addOccasion()
Order		sendOrder() editOrder() cancelOrder()

Table 2

### **3.6 Global Software Control**

The type of global control flow used here is 'Event driven'. The system works on the services requested by the *Manager*. As a result the system waits for some activity on the part of the *Manager* to initiate any process. Until then the system waits on the main user interface that provides the manager access to various functions on the system.

To ensure a robust design of the system, we define some strategic goals for the system as mentioned below:

1. *Boundary objects do not define any of the fields in the System, instead they are only associated with creation of control objects upon request of access to specific functions by the Manager.*
2. *Control Objects must not be shared among functions, instead if one function on the interface is activated the other functions must not be accessible by the Manager until the current function complete its operation. This will avoid any other control object being made by the user when some other control object exists.*
3. *Entity objects must not allow direct access to its fields to any other class or object, instead it should use getter and setter method to get and set the values of its attributes for better encapsulation. Also, the attributes of the class must be declared private for access control and avoiding accidental changes by other objects.*
4. *Database connection must not be open all the time, instead the connection to the database should be made only when any functions requests data or wishes to update data. This will ensure that the data is not manipulated by the system without any explicit requests made.*
5. *Entity data validation should be conducted at point where data is written into the database, this will ensure that no invalid data is entered into the database avoiding any serious inventory slips in the future.*
6. *The system will require an authenticated login and password for accessing the main interface and hence all the functions, this will prevent any unauthorized login into the system and hence make the system secure. The master login and password will also ensure that the user is enabled to connect to the database subsystem with any explicit login into the database management system.*

### **3.7 Boundary Conditions**

*StartSystem*: The *Manager* initiates the system using this function. At the startup, the *Manager* is asked for authentication and if the authentication is successful, the main interface becomes visible to the manager. There are no database connections established at in this phase hence meeting our global control flow specification mentioned earlier. The *Manager* can now access and perform various services accessible from the main interface. Any function accessed opens a database connection to the desired database and closes the connection on termination of the function.

*ExitSystem*: The *Manager* can stop the system using the exit function on the main interface. This action terminates the system. It is assumed that there are no active database connections at this point of time as the *Manager* is at the main interface windows of the system and the design goals define no database connectivity at this instance. Database connections are opened and closed at the initiation and termination of the certain

#### **Defining Exceptions**

The scenarios of failure of the system can be stated as follows:

- The database connection cannot be established.
- Incorrect data is entered into the database in spite of the validations being carried out.
- The system crashes in the middle of an update process being carried out.

To deal with the above mentioned exceptions, we define two use-cases that will be used to overcome or at least reduce the effects of the exception.

CheckDataIntegrity	This use case can be invoked in the event of the one of the last two exceptions occurs. This use-case will run through the whole database and check for non-related data entries, incorrect data entries and incomplete data-entries. It will then delete these irrelevant data entries from the database and provide the <i>Manager</i> with an error free database to work with.
ResetDataConnectivity	This use case can be invoked in the event the first exception occurs. This use case with restart/re-establish all the database connectivity for the application, invoke the CheckDataIntegrity use case mentioned above and then provide the user with fresh error free database connectivity.

Table 3

## 4. Subsystem services

The subsystem services diagram can be shown below

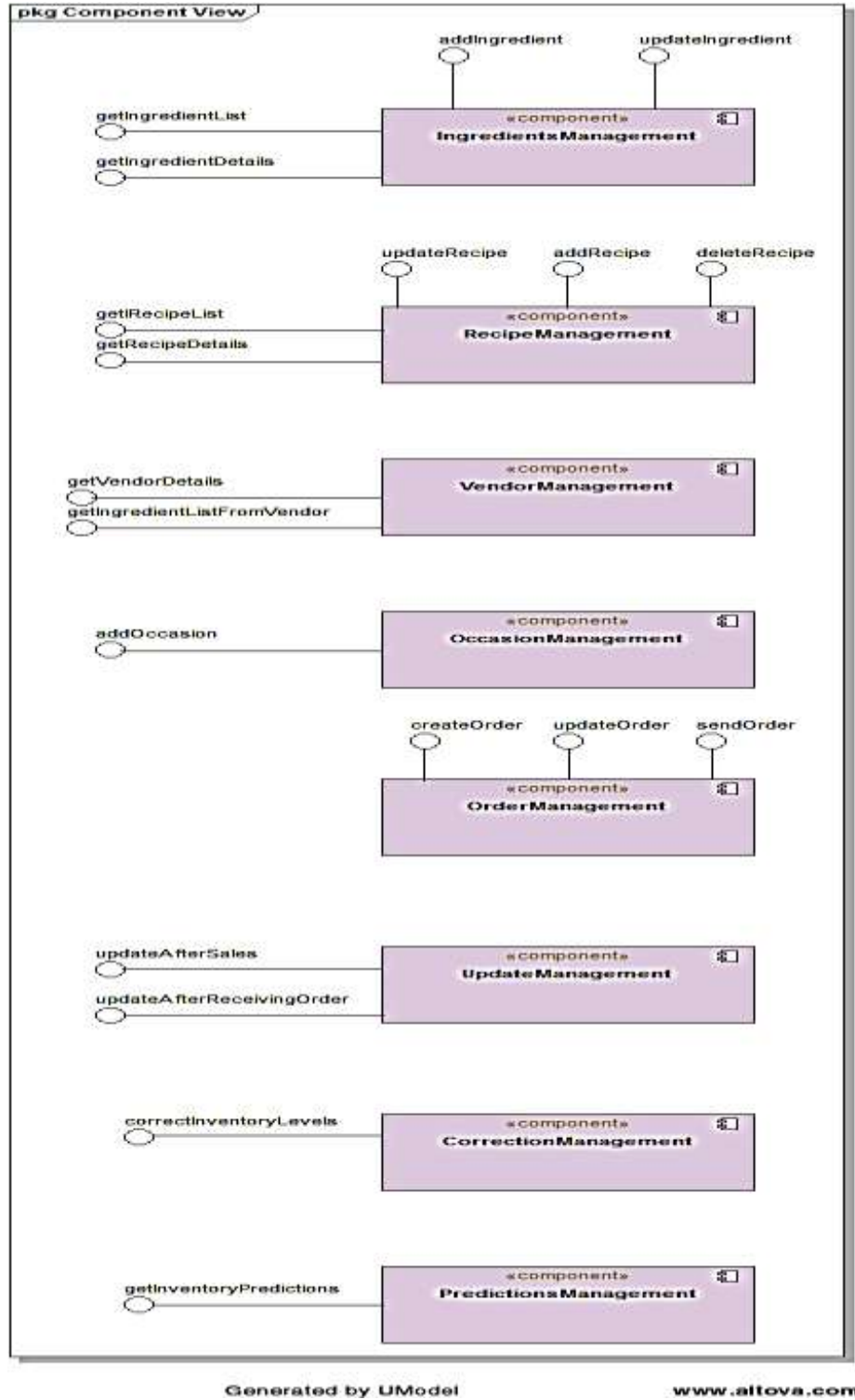


Figure 3

## 5. Database System

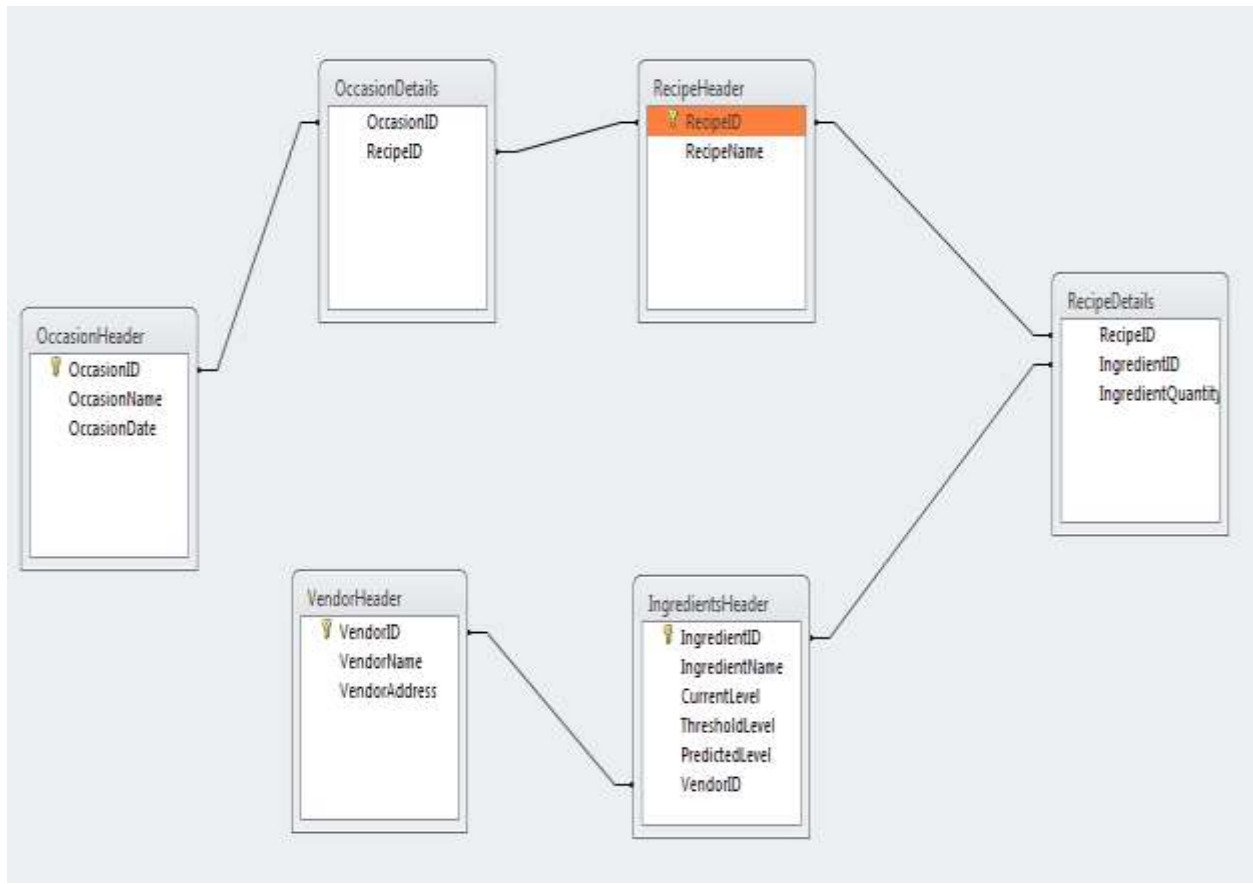


Figure 4

The above figure displays the database schema of the system. The total inventory is stored and maintained using this schema of the database. The schema contains six tables with relationships between them as shown. The entries in the boxes represent the column names. The primary keys are unique and any relationship line from the primary key to a non-primary entity is one to many relationships.



## **6. Object Design Tradeoffs**

- **Space vs. Speed:** The product is based on data inventory management and thus requires a lot of space for the storage of data. Thus to make the system read and write data at a faster speed we need more memory space.
- **Build vs. Purchase:** This product mostly uses open source products as development tools, so the scope of purchase is minimized of off the shelf products required for this product. Also this product uses open source libraries and source codes for some parts thus avoiding the Build vs. Buy dilemma.
- **Delivery time vs. Functionality:** As this project is running on a tight schedule, it will be difficult to produce a system with all the functionalities that are mentioned in the specifications by the date of delivery but a prototype system with the important functionality must be ready by the requested date. Functionalities such as prediction of data can be delivered at a later time.

## 7. Interface Documentation Guidelines

Below are mentioned guidelines for the interface documentation:

- The names of the class should start with an uppercase letter and if the class name consists of more than one word then *CamelCasing* should be used.

Example: Public class Ingredient{ }

- The class methods begin with a lowercase and if the method name consists of more than one word, *camelCasing* should be used.

Example: public boolean addIngredient()

- The class attributes start with uppercase letter and for attributes with multiple words, CamelCasing must be used.

Example: int IngredientID

- The Package names should start with an uppercase letter and use CamelCasing when package names consist of multiple names. Also the package names must end with the word 'Package'

Example: package *IngredientsPackage*

- Constants are represented by all uppercase letters and constant names with multiple words should be separated with underscores ('\_').

Example: TOTAL\_NUMBER\_OF\_RECIPES

- The class diagrams represent the access specifiers using symbols that can be summarized in the image shown below

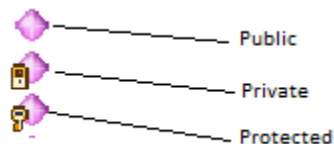


Figure 5

- The method and attribute names should be such that they are self-explanatory of their context of use.

Example: If the method adds a new recipe to the database the name of the method should be *addRecipe*.

If an attribute holds the value of the name of the ingredient, the name of the attribute should be *IngredientName*.

- Comments must be used extensively to make the code easily understandable. All the classes must have preface comments, all the methods must have the functionality commented in the code and the attributes must have their usage comments too.

## 8. Packages

The figure below gives an overview of the package composition of the system.

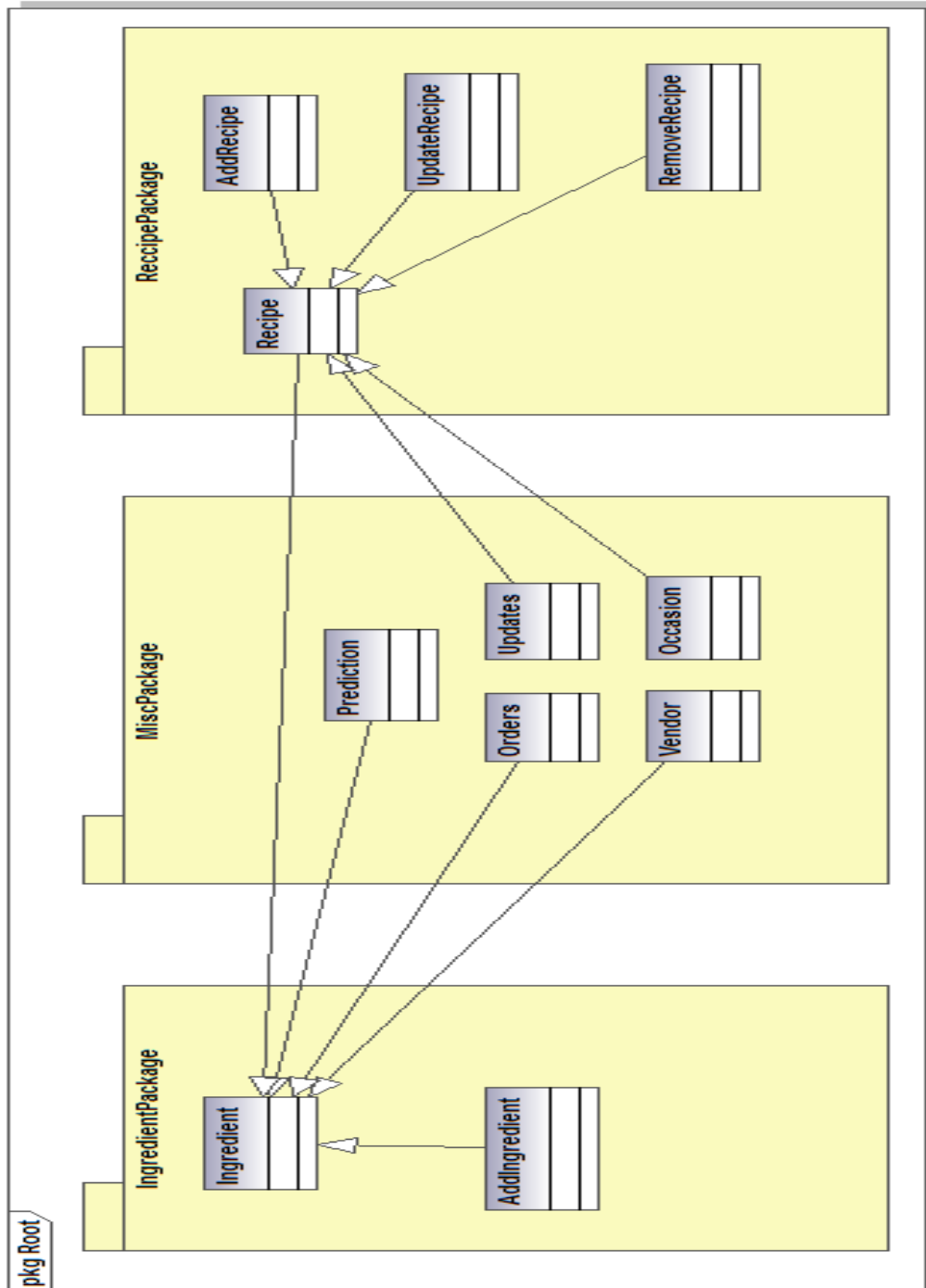


Figure 6

## **Package Description**

**IngredientPackage:** This package constitutes of two classes namely the Ingredient and Add Ingredient which forms the basis of the system. This package provides functionalities to the user for adding new ingredients in the database. It also provides the current list of ingredients in the database and the corresponding details. These functions act as a basic functionality for the classes that inherit the classes in the IngredientPackage. For instance, the Recipe class requires the list of ingredients to assign ingredients to the recipe that is being added, the list of ingredient is required for creating orders for the corresponding recipe. Thus in short this package has classes that provide vital functionality to the classes in the other two packages.

**MiscPackage:** This package constitutes of five classes which perform tasks quite different to each other and are linked to the other two packages by the connections differing in functionality. The classes namely are Prediction, Orders, Updates, Vendor and Occasion. The Prediction and Occasion classes can be considered as a special feature wherein ingredients are constantly being used and if any occasion is near, then the prediction is done in accordance with the existing levels and past history of usage. The Vendor and Orders are intertwined amongst themselves as in the usage. If low inventory levels are sensed then the orders of required ingredients are passed by to the manager and an order form is generated which is given to the vendor for the replenishing the inventory. Once the stock levels are more than the threshold level then the update can be performed and the new levels are taken into consideration.

**RecipePackage:** This Package Constitutes the Recipe, AddRecipe and RemoveRecipe which forms its classes. The AddRecipe classes is usually used so as to include a new row in the recipe table and which in turn is linked to the ingredients as when the following is utilized it indirectly uses up the ingredients involved. So along with the recipe, all the links to the ingredient are mandatory. Removing the recipe from the recipe list may not affect the ingredients as the one to many relation for the recipe and ingredients is still preserved. This package acts as an interface for the user, usage in order to make changes into the inventory with perspective to usage. The recipe details usually include the recipe name, recipe ID and the associated Ingredient ID as well. These classes mentioned above are inter linked so as to form a cohesive output.

## 9. Class Interfaces

The figure below shows the class diagram of the whole system.

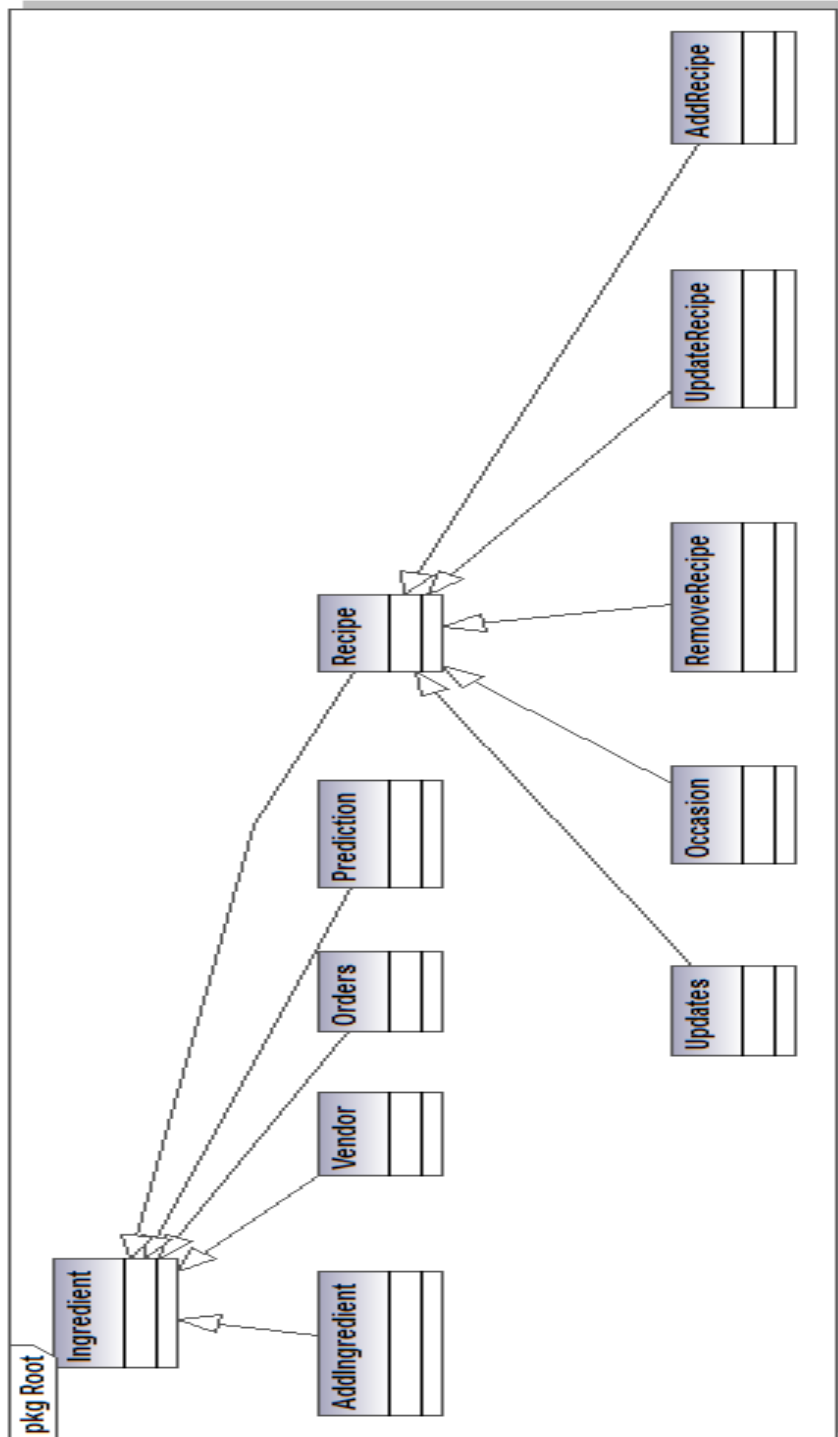
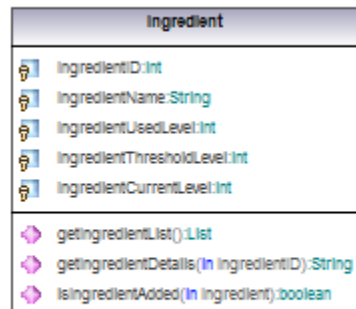


Figure 7

The figure above gives an overview of the class structure of the system. The details of the attributes and functions along with access specifiers, return types and parameters are listed in the diagrams below.

- **Class Ingredient**



Generated by UModel

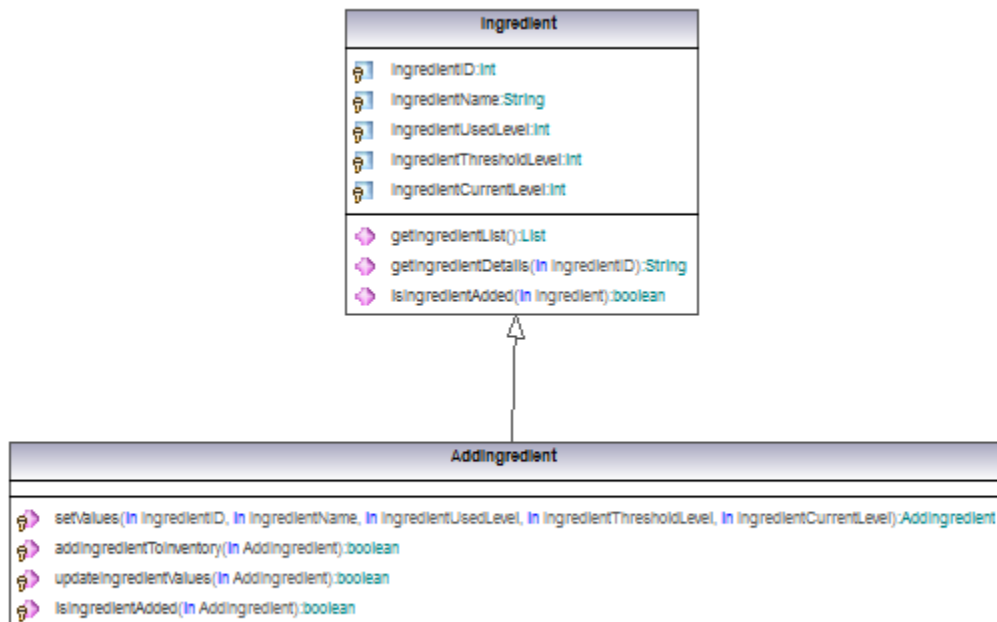
[www.altova.com](http://www.altova.com)

Figure 8

### **Contracts for class Ingredient**

- a. Context Ingredient :: getIngredientList() **pre:**  
!isIngredientAdded(i : Ingredient)
- b. Context Ingredient :: getIngredientDetails(IngredientsID) **pre:**  
!isIngredientAdded(i:Ingredient)

- **Class AddIngredient**



Generated by UModel

[www.altova.com](http://www.altova.com)

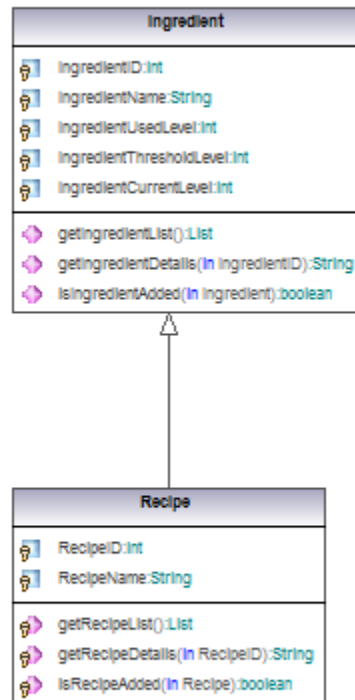
Figure 9

**Contracts for class AddIngredient**

- Context AddIngredient :: addIngredientToInventory(a:AddIngredient) **pre:**  
!isIngredientAdded(a)
- Context AddIngredient :: updateIngredientValues(a:AddIngredient) **pre:**  
!isIngredientAdded(a)
- Context AddIngredient :: addIngredientToInventory (a:AddIngredient) **post:**  
isIngredientAdded(a)
- Context AddIngredient :: updateIngredientValues(a:AddIngredient) **post:**  
isIngredientAdded(a)



- **Class Recipe**



Generated by UModel

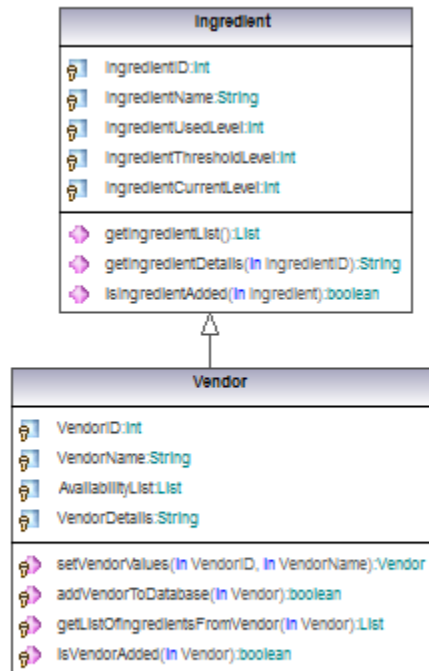
[www.altova.com](http://www.altova.com)

Figure 10

### Contracts for class Recipe

- a. Context Recipe :: getRecipeList() **pre:**  
isRecipeAdded(Recipe)
- b. Context Recipe :: getRecipeDetails(r : Recipe) **pre:**  
isRecipeAdded(r)

- **Class Vendor**



Generated by UModel

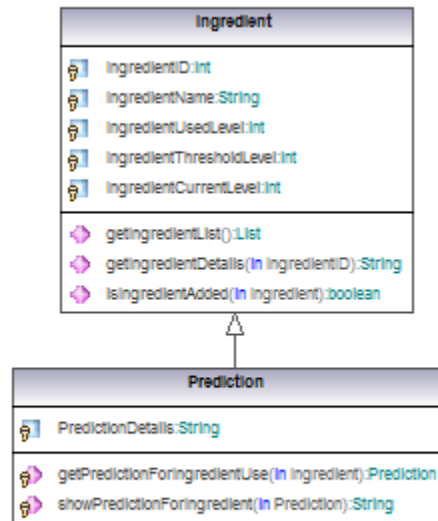
[www.altova.com](http://www.altova.com)

Figure 11

**Contracts for class Vendor**

- Context Vendor :: addVendorToDatabase(v:Vendor) **pre:**  
!isVendorAdded(v)
- Context Vendor :: addVendorToDatabase(v:Vendor) **post:**  
isVendorAdded(v)
- Context Vendor :: getListOfIngredientsFromVendor(v:Vendor) **pre:**  
isVendorAdded(v)

- **Class Prediction**

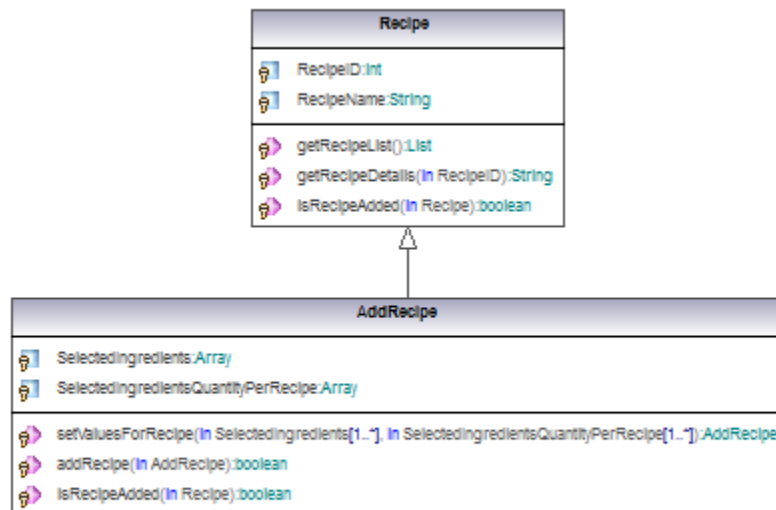


Generated by UModel

[www.altova.com](http://www.altova.com)

Figure 12

- **Class AddRecipe**



Generated by UModel

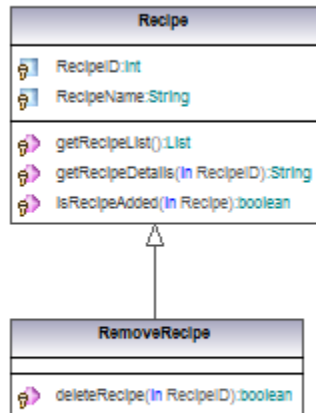
[www.altova.com](http://www.altova.com)

Figure 13

### Contracts for class AddRecipe

- Context AddRecipe :: addRecipe(r : AddRecipe) **pre:**  
!isRecipeAdded(r)
- Context AddRecipe :: addRecipe(r : AddRecipe) **post:**  
isRecipeAdded(r)

- **Class RemoveRecipe**



Generated by UModel

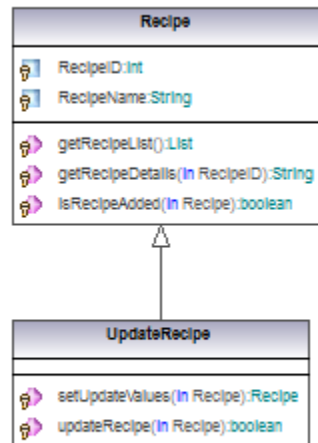
[www.altova.com](http://www.altova.com)

Figure 14

### Contracts for class RemoveRecipe

- a. Context `RemoveRecipe :: deleteRecipe(r : Recipe)` **pre:**  
`isRecipeAdded(r)`
- b. Context `RemoveRecipe :: deleteRecipe(r : Recipe)` **post:**  
`!isRecipeAdded(r)`

- **Class UpdateRecipe**



Generated by UModel

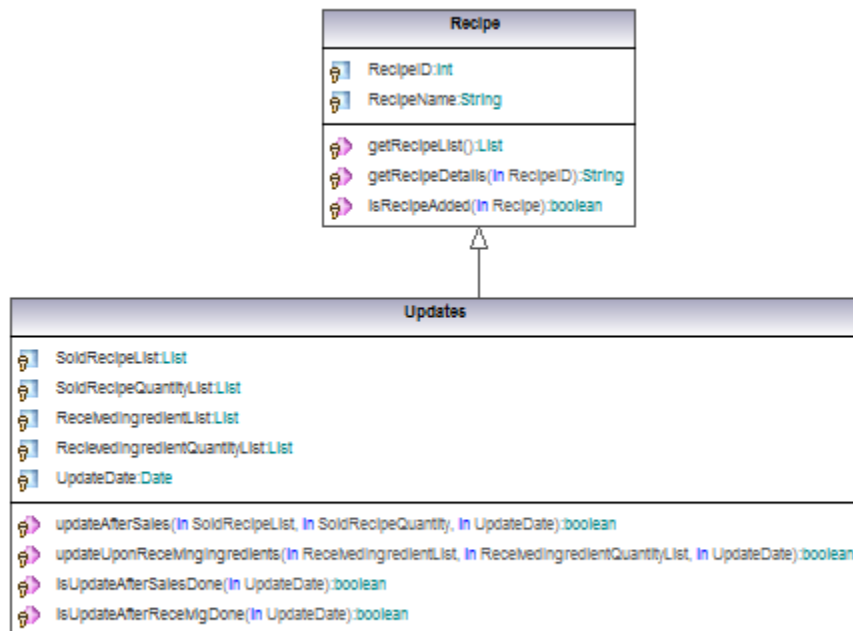
[www.altova.com](http://www.altova.com)

Figure 15

### Contracts for class UpdateRecipe

- a. Context UpdateRecipe :: updateRecipe(r : Recipe) **pre:**  
isRecipeAdded(r)
- b. Context UpdateRecipe :: updateRecipe(r : Recipe) **post:**  
isRecipeAdded(r)

- **Class Updates**



Generated by UModel

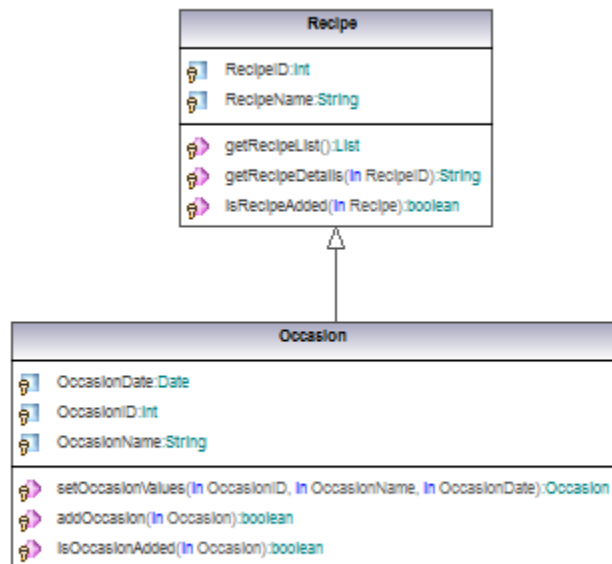
[www.altova.com](http://www.altova.com)

Figure 16

### Contracts for class Updates

- Context Updates :: updateAfterSales(srl: SoldRecipeList, srq: SoldRecipeQuantity, up: UpdateDate) **pre:**  
!isUpdateAfterSalesDone(up)
- Context Updates :: updateAfterSales (srl: SoldRecipeList, srq: SoldRecipeQuantity, up: UpdateDate) **post:**  
isUpdateAfterSalesDone(up)
- Context Updates :: updateUponReceivingIngredients(ril: ReceivedIngredientList, riq: ReceivedIngredientQuantity, up: UpdateDate) **pre:**  
!isUpdateAfterSalesDone(up)
- Context Updates :: updateUponReceivingIngredients(ril: ReceivedIngredientList, riq: ReceivedIngredientQuantity, up: UpdateDate) **post:**  
isUpdateAfterSalesDone(up)

- **Class Occasion**



Generated by UModel

[www.altova.com](http://www.altova.com)

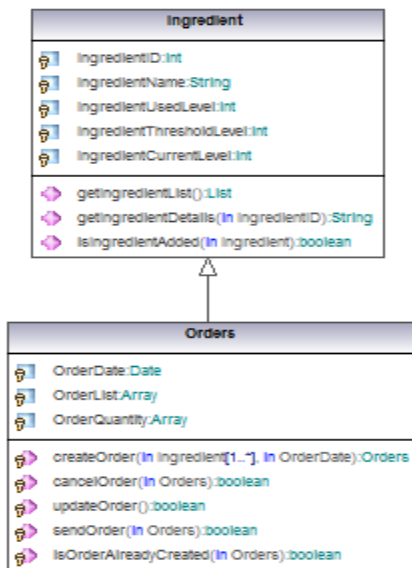
Figure 17

### Contracts for class Occasion

- a. Context Occasion :: addOccasion(oc : Occasion) **pre:**  
!isOccasionAdded(oc)
- b. Context Occasion :: addOccasion(oc : Occasion) **post:**  
isOccasionAdded(oc)



- **Class Orders**



Generated by UModel

[www.altova.com](http://www.altova.com)

Figure 18

### Contracts for class Orders

- Context Orders :: createOrder(i:Ingredient, o:OrderDate) **pre:**  
!isOrderAlreadyCreated(i,o)
- Context Orders :: createOrder(i:Ingredient, o:OrderDate) **post:**  
isOrderAlreadyCreated(i,o)
- Context Orders :: cancelOrder(o:Orders) **pre:**  
isOrderAlreadyCreated(o)
- Context Orders :: updateOrder(o:Orders) **pre:**  
isOrderAlreadyCreated (o)

## **Bibliographical References**

- Bernd Bruegge & Allen H. Dutoit. Object-Oriented Software Engineering Using UML, Patterns, and Java, Third Edition, 2005.
- Object design tradeoffs for Project, Anonymous.  
<http://www.scribd.com/doc/38302828/29/Object-design-trade-offs>
- Technical Design Document. [www.in.gov/fssa/files/QualCheck.pdf](http://www.in.gov/fssa/files/QualCheck.pdf)
- Union Design Pattern: Inheritance and Polymorphism. <http://cnx.org/content/m11796/latest/>

## Glossary

<b>Manager</b>	A manager is the person who is here considered to be the person in charge. He undertakes responsibility of maintaining, operating the store and keeps information of each and every specifics. He is also responsible for taking decisions regarding issuing orders to vendors , add new vendors, create new recipes ,etc.
<b>Recipe</b>	A recipe is a dish that is made using the ingredients from the store. The recipes and ingredients are inter-related. A recipe is usually made up of two or more ingredients.
<b>Ingredients</b>	Ingredients here considered to be atomic substances which are used as a component of a recipe. These basic ingredients cluster to form a recipe. The availability of these help in creating new recipes and regularly checking the inventory levels and prediction analysis.
<b>Vendor</b>	The necessary consumable items (ingredients) which are used up in preparing the recipes are provided by the vendor on a regular basis. The manager is responsible for checking the current level of inventory with the threshold levels and if any particular ingredient is found to be in the danger zone then an order is processed by the manager to the vendor.
<b>Order</b>	Order is a generalized function used for replenishment of used up ingredients. The order usually consists of an order form which is supplied by the manager to the vendors.
<b>Add Recipe</b>	This action leads to the addition of a new recipe into the existing list of recipes. When a new recipe is made it also links up to the ingredients being used. So whenever a particular recipe is ordered it ends up using its required ingredients.
<b>Remove Recipe</b>	The manager initiates the remove the recipe action. Once a recipe is removed it is also cleared from the RECIPE table.
<b>Add Vendor</b>	Add vendor is the case when a new ingredient is being added to the database and the present vendor isn't supplying that particular item.
<b>Remove Vendor</b>	If a vendor is removed from the list of vendors then the recipe table reflects only the recipes that are not linked with that particular vendor.
<b>Check Threshold</b>	Check threshold level refers to checking the inventory levels of all the ingredients. This provides information pertaining to the next order (vendor) and also the recipe selection.
<b>Process Order</b>	A process order issues a purchase/required list of ingredients and accordingly generates a purchase order for the vendor.
<b>Update Resource Database</b>	Updating the resource gives a crystal clear idea to the manager regarding the usage of the inventory from the quantity sold in a particular time period. This leads to the checking the threshold values.
<b>Add Occasion</b>	Occasion refers to something special and this leads to extra needs from the vendor. So accordingly when an occasion is added the specials recipes are kept in mind and a purchase order to satisfying the special needs is generated.