Assignment - Design Pattern

Version: February 3, 2022

Objectives

Use design patterns to build the application

General (Note the use of REQUIREMENTS 😊)

1. You MUST create 5 design patterns for the application specified below (each 8 points).
2. You MUST use Java!
3. You MAY use Astah to generate the skeleton Java code from the UML (Unified Modeling Language). You might want to incorporate the "Class Description" tables into your UML before generating code. If you do, add the UML files to the project upload (with your name in a comment block).
4. You do NOT:
   a) Need to meet all the functional requirements specified in Task 1 but the general idea should be visible
   b) Must have a running/working application for all features (but it should not have compiler errors or warnings!).
5. We will grade you based on the code that you have written.
6. Please indicate (as comments in submission box on Canvas) which design patterns you implemented and the class file or piece of code where it is implemented (this documentation is important and will be graded!).

As stated before, you do not have to implement all the requirements and for 1. Façade you can choose any 4 design patterns listed below in "Introducing Design Patterns" that you deem fit. The HINTS might help you. You are relatively free of how to do things. Please mark where you see your Design Patterns in your code. **This is coding intensive! Start early it will take a while to figure things out.**

Introducing Design Patterns (40)

In this homework, we implement five design patterns into the Product Trading and Bidding System (**PTBS**). The five design patterns, which are implemented within the PTBS systems, are **Façade, Bridge, Factory Method, Iterator, and Visitor**. In the following sections, a brief description of the patterns and the detailed implementation of the patterns are presented.

1. Façade

The façade pattern can make the task of accessing many modules much simpler by providing an additional interface layer. In the implementation of the PTBS system, the façade lies in the top of all the interfaces and modules. The façade object provides a single interface to the more general facilities of other subsystems. The benefits offered by façade are as follows.

1. It shields the subsystem components to the PTBS system. For example, in the PTBS system, the main function does not have to deal with all the subsystem components. Instead, it just passes the control to the façade object and the façade object wraps up all the subsystem components.

2. The façade object eliminates the dependencies between objects.

Figure 1 shows the structure of the façade object and its subsystem classes. Table 1 shows the class description of façade object.

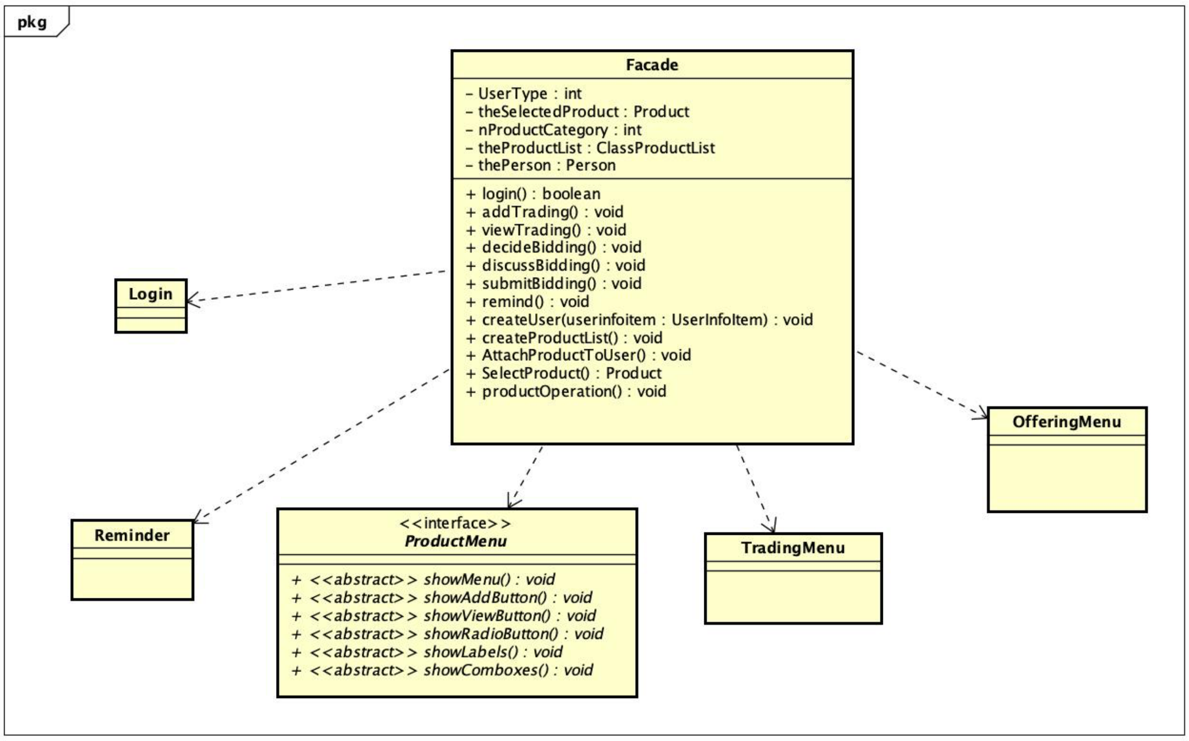
pkg
Facade
- UserType : int
- theSelectedProduct : Product
- nProductCategory : int
- theProductList : ClassProductList
- thePerson : Person
+ login() : boolean
+ addTrading() : void
+ viewTrading() : void
+ decideBidding() : void
+ discussBidding() : void
+ submitBidding() : void
+ remind() : void
+ createUser(userinfoitem : UserInfoItem) : void
+ createProductList() : void
+ AttachProductToUser() : void
+ SelectProduct() : Product
+ productOperation() : void
Login
OfferingMenu
Reminder
<<interface>>
ProductMenu
+ <<abstract>> showMenu() : void
+ <<abstract>> showAddButton() : void
+ <<abstract>> showViewButton() : void
+ <<abstract>> showRadioButton() : void
+ <<abstract>> showLabels() : void
+ <<abstract>> showComboxes() : void
TradingMenu

Figure 1 Façade Structure

Table 1 Class description of façade

Class Name	Façade	
Descriptions	The interface class between the GUI and the underlining system, the control logic and many of the operating functions are included in this class	
Attribute	Name	UserType
	Des	The type of the user: Buyer: 0, Seller 1
Attribute	Name	theSelectProduct
	Des	The object that holds the currently selected product.
Attribute	Name	nProductCategory
	Des	The selected product category: 0: Meat, 1: Produce.
Attribute	Name	theProductList
	Des	The list of products of the entire system.
Attribute	Name	thePerson
	Des	The current user.
Function	Name	login
	Des	Show login GUI and return the login result.
Function	Name	addTrading

	Des	When clicking the add button of the ProductMenu, call this function. This function will add a new trade and fill in the required information. This function will be called SellerTradingMenu or BuyerTradingMenu based on the type of the user. It will not update the product menu. The product menu needs to be refreshed outside the function.
Function	Name	viewTrading
	Des	When clicking the view button of the ProductMenu, call this function and pass the pointer of the Trading and the person pointer to this function. This function will view the trading information. This function will call SellerTradingMenu or BuyerTradingMenu according to the type of the user.
Function	Name	viewOffering
	Des	This function will view the given offering.
Function	Name	markOffering
	Des	Set the deal flag of the given offering.
Function	Name	submitOffering
	Des	Used by the buyer to submit the offering.
Function	Name	remind
	Des	Show the remind box to remind buyer of the upcoming overdue trading window.
Function	Name	createUser(UserInformation userinfoitem)
	Des	Create a user object according to the userinfoitem, the object can be a buyer or a seller.
Function	Name	createProductList()
	Des	Create the product list of the entire system.
Function	Name	attachProductToUser
	Des	Call this function after creating the user. Create productList by reading the UserProduct.txt file. Match the product name with theProductList. Attach the matched product object to the new create user: Facade.thePerson. ProductList
Function	Name	selectProduct
	Des	Show the Product list in a Dialog and return the selected product.
Function	Name	productOperation
	Des	This function will call the thePerson. CreateProductMenu0 According to the real object (buyer or seller) and the productLevel, it will call different menu creator and show the menu differently according to the usertype.

Bridge

The **Bridge** pattern affects the load menu option in the PTBS system. When a user logs in, the bridge will help to load the appropriate menu for either buyer or seller. Furthermore, the menu should be different depending on which kind of product is selected, i.e., meat or produce product. This feature is implemented by bridge pattern.

In the bridge that is implemented in PTBS, the ProductMenu is the implementor class hierarchy, which has two subclasses named MeatProductMenu and ProduceProductMenu, and the Person is the abstraction class hierarchy which has two subclasses named Seller and Buyer.

The benefits of bridge pattern are that the implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at runtime. For example, the 'real' user interface of the ProductMenu is configured at runtime depends on the type of the product (meat product or produce product) and the type of the user (buyer or seller). And it is easy to extend the functionality that could be done by the load menu option. For example, if one other type of user or product is added, we can extend the abstraction (person) or implementor (productMenu) accordingly and independently. Without a Bridge, we would have to instantiate 4 subclasses for two kinds of user and two kinds of products. Even worse, if we add a new type of product, we will have to add two subclasses for both seller and buyer. Figure 2 illustrates the structure of the Bridge pattern in this system. Table 2 shows the class description of Bridge pattern in this implementation.

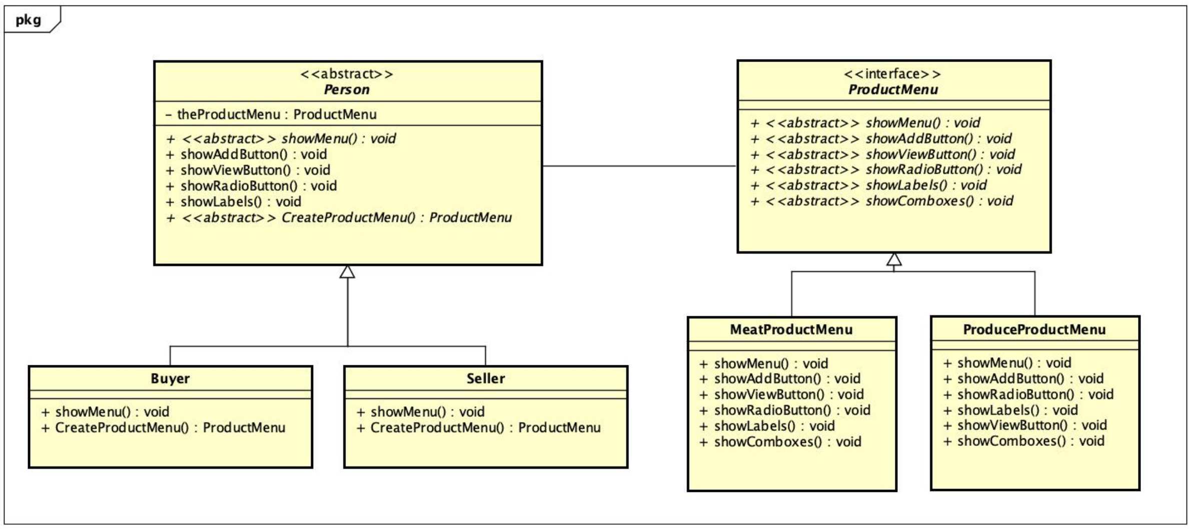
pkg
<<abstract>>
Person
- theProductMenu : ProductMenu
+ <<abstract>> showMenu() : void
+ showAddButton() : void
+ showViewButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ <<abstract>> CreateProductMenu() : ProductMenu
<<interface>>
ProductMenu
+ <<abstract>> showMenu() : void
+ <<abstract>> showAddButton() : void
+ <<abstract>> showViewButton() : void
+ <<abstract>> showRadioButton() : void
+ <<abstract>> showLabels() : void
+ <<abstract>> showComboxes() : void
Buyer
+ showMenu() : void
+ CreateProductMenu() : ProductMenu
Seller
+ showMenu() : void
+ CreateProductMenu() : ProductMenu
MeatProductMenu
+ showMenu() : void
+ showAddButton() : void
+ showViewButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ showComboxes() : void
ProduceProductMenu
+ showMenu() : void
+ showAddButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ showViewButton() : void
+ showComboxes() : void

Figure 2 Bridge Structure

Table 2 Class description of Bridge

Class Name	Abstract Person	
Descriptions	The abstract class on one side of the bridge.	
	Name	theProductMenu
Attribute	Des	Variable of ProductMenu. Use this variable to point to a concrete productMenu object. Later, it will operate the object.
	Name	showAddButton
Function	Des	Call the implementation to show the "add" buttons.
Function	Name	showViewButton

	Des	Call the implementation to show the "view" buttons.
Function	Name	showRadioButton
	Des	Call the implementation to show the radio buttons.
Function	Name	showLabels
	Des	Call the implementation to show the labels.
Abstract Function	Name	showMenu
	Des	Overridden by the class: buyer and seller to show the menu

Class Name	Buyer	
Descriptions	The concrete subclass of Person.	
Function	Name	showMenu
	Des	According to the need of buyer show the appropriate items on the menu.

Class Name	Seller	
Descriptions	The concrete subclass of Person.	
Function	Name	showMenu
	Des	According to the need of seller show the appropriate items on the menu.

Class Name	abstract ProductMenu	
Descriptions	The abstract class on the other side of the bridge.	
Abstract Function	Name	showAddButton
	Des	To show the add buttons.
Abstract Function	Name	showViewButton
	Des	To show the view buttons.
Abstract Function	Name	showRadioButton
	Des	To show the radio buttons.
Abstract Function	Name	showLabels
	Des	To show the labels.

Class Name	MeatProductMenu	
Descriptions	One concrete implementation of ProductMenu for the meat product.	
Function	Name	showAddButton
	Des	To show the add buttons.
Function	Name	showViewButton
	Des	To show the view buttons.
Function	Name	showRadioButton
	Des	To show the radio buttons.

Function	Name	showLabels
	Des	To show the labels.

Class Name	ProduceProductMenu	
Descriptions	One concrete implementation of ProductMenu for the produce product.	
Function	Name	showAddButton
	Des	To show the add buttons.
Function	Name	showViewButton
	Des	To show the view buttons.
Function	Name	showRadioButton
	Des	To show the radio buttons.
Function	Name	showLabels
	Des	To show the labels.

Factory Method

The **Factory Method** pattern enables the subclasses to decide which class to instantiate. In the PTBS system, the Factory Method pattern is implemented when the ProductMenu is loaded. The ProductMenu depends on product category and user type. The Factory Method will determine which class to instantiate.

The benefit of the Factory Method is to eliminate the need to bind application-specific classes into the code. The code only deals with the Product interface (ProductMenu in this system), so it can deal with any ConcreteProduct class (MeatlProductMenu and ProduceProductMenu in this system). Figure 3 illustrates the structure of the Factory Method implemented in this system. Table 3 shows the class description of the Factory Method.

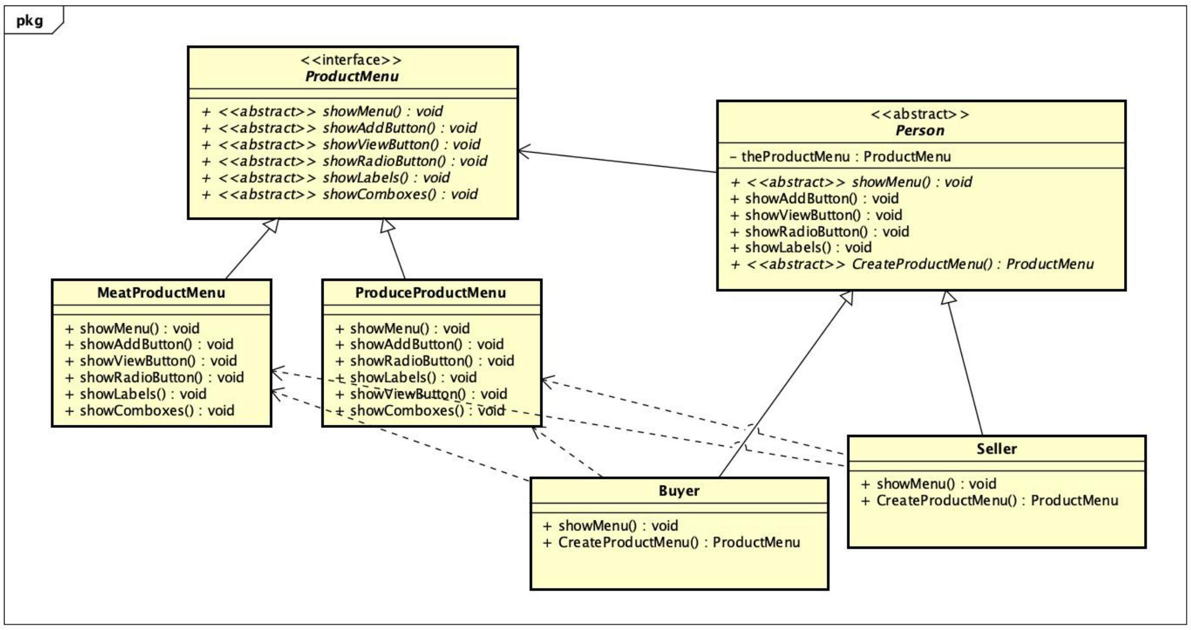
pkg
<<interface>>
ProductMenu
+ <<abstract>> showMenu() : void
+ <<abstract>> showAddButton() : void
+ <<abstract>> showViewButton() : void
+ <<abstract>> showRadioButton() : void
+ <<abstract>> showLabels() : void
+ <<abstract>> showComboxes() : void
<<abstract>>
Person
− theProductMenu : ProductMenu
+ <<abstract>> showMenu() : void
+ showAddButton() : void
+ showViewButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ <<abstract>> CreateProductMenu() : ProductMenu
MeatProductMenu
+ showMenu() : void
+ showAddButton() : void
+ showViewButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ showComboxes() : void
ProduceProductMenu
+ showMenu() : void
+ showAddButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ showViewButton() : void
+ showComboxes() : void
Seller
+ showMenu() : void
+ CreateProductMenu() : ProductMenu
Buyer
+ showMenu() : void
+ CreateProductMenu() : ProductMenu
Figure 3 Factory method structure

Table 3 Class description of factory method

Class Name	Person	
Descriptions	The Person class is involved in the bridge pattern to show the Menu, and in factory menu to create proper menu object. It has no idea of the concrete high-level or low-level menu.	
Abstract Function	Name	createProductMenu()
	Des	The abstract factory method.

Class Name	Seller	
Descriptions	The concrete implementation of the Person class.	
Function	Name	createProductMenu
	Des	According to the Product type create a concrete product menu: meat or produce.

Class Name	Buyer	
Descriptions	The concrete implementation of the Person class.	
Function	Name	createProductMenu
	Des	According to the Product type create a concrete product menu: meat or produce.

Class Name	abstract productMenu
Descriptions	The abstract product of the factor method.

Class Name	MeatProductMenu
Descriptions	A subclass of ProductMenu. One of the concrete products of the factor method.

Class Name	ProduceProductMenu
Descriptions	A subclass of ProductMenu. One of the concrete products of the factor method.

Iterator

The **Iterator** pattern is implemented as a means for distributing the grade report for the buyers. The Iterator class defines an interface for accessing the list's elements without exposing its internal structure. The OfferingIterator is implemented as an Iterator to the OfferingList class. It simplifies the Aggregate interface and supports variations in the traversal of an aggregate. Figure 4 illustrates the structure of the Iterator Pattern in the PTBS system. Table 4 lists the class description of the Iterator pattern.

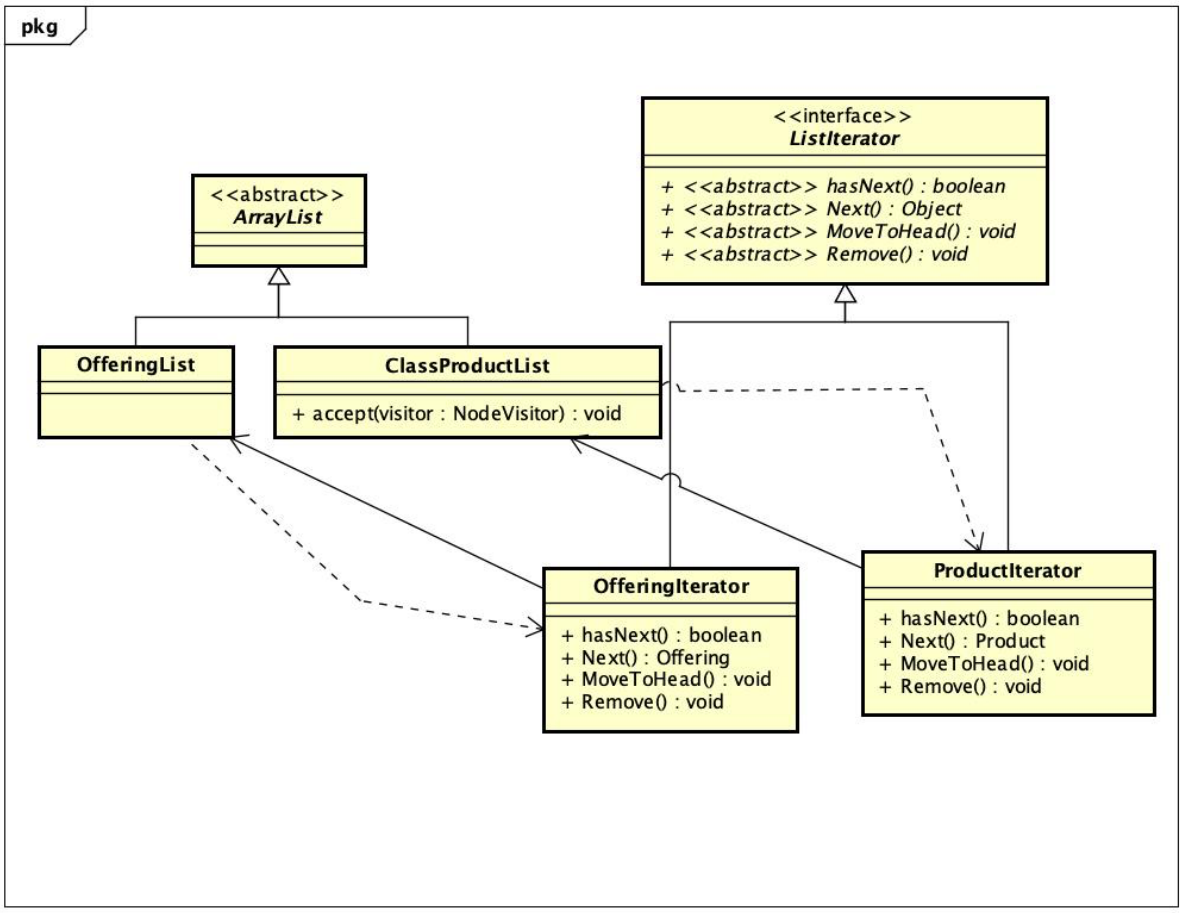

Figure 4 Iterator structure

Table 4 Class description of iterator

Class Name	ArrayList
Descriptions	The abstract class of the list to be iterated. [Built into Java]

Class Name	OfferingList
Descriptions	Subclass of ArrayList. One concreted List class that needs to be iterated.

Class Name	*ClassProductList*
Descriptions	The abstract class of the list to be iterated

Class Name	ListIterator	
Descriptions	The abstract iterator class in this class declares some iterating functions that need to be implemented in the concrete iterator classes.	
Abstract Function	Name	hasNext
	Des	If in the iterator there exists the "next", return true; else return false.

Abstract Function	Name	next
	Des	If hasNext, return the next Item, move the current Item to the next item. Else return null.
Abstract Function	Name	remove
	Des	Remove the current item from the list.
Abstract Function	Name	moveToHead
	Des	Set the current item to the location before the first item.

Class Name	OfferingIterator	
Descriptions	A concrete subclass of ListIterator that iterate the OfferingList	
Function	Name	hasNext
	Des	If in the OfferingIterator there exists the "next", return true; else return false.
Function	Name	next
	Des	If hasNext, return the next offering, move the current Item to the next offering. Else return null.
Function	Name	remove
	Des	Remove the current offering from the list.
Function	Name	moveToHead
	Des	Set the current offering to the location before the first offering.

Class Name	ProductIterator	
Descriptions	A concrete subclass of ListIterator that iterates the ProductList	
Function	Name	hasNext
	Des	If in the ProductIterator there exists the "next", return true; else return false.
Function	Name	next
	Des	If hasNext, return the next product, move the current Item to the next product. Else return null.
Function	Name	remove
	Des	Remove the current product from the list.
Function	Name	moveToHead
	Des	Set the current product to the location before the first product.

Visitor

The purpose of the **Visitor** Pattern is to encapsulate an operation that you want to perform on the elements of a data structure. In this way, you can change the operation being performed on a structure without the need to change the classes of the elements that you are operating on. Using a Visitor pattern allows you to decouple the classes for the data structure and the algorithms used upon them.

The benefit of Visitor is that Visitor makes adding new operation easy.

Each node in the data structure "accepts" a Visitor, which sends a message to the Visitor, which includes the node's class. The visitor will then execute its algorithm for that element. In our implementation of the PTBS system, ReminderVisitor provides the Visitor capacity. Figure 5 illustrates the structure of the Visitor Pattern in PTBS system. Table 5 lists the class description of the Visitor Pattern.

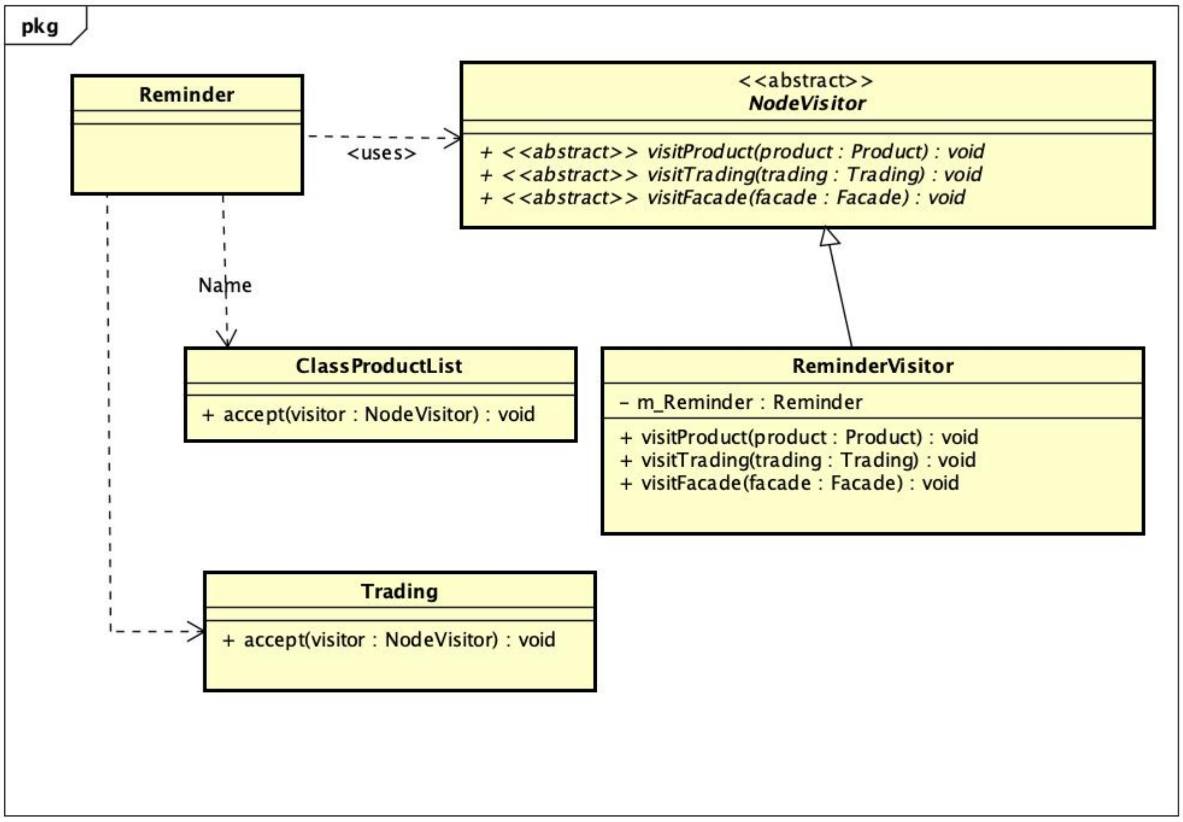
pkg
Reminder
<uses>
<<abstract>>
NodeVisitor
+ <<abstract>> visitProduct(product : Product) : void
+ <<abstract>> visitTrading(trading : Trading) : void
+ <<abstract>> visitFacade(facade : Facade) : void
Name
ClassProductList
+ accept(visitor : NodeVisitor) : void
ReminderVisitor
- m_Reminder : Reminder
+ visitProduct(product : Product) : void
+ visitTrading(trading : Trading) : void
+ visitFacade(facade : Facade) : void
Trading
+ accept(visitor : NodeVisitor) : void

Figure 5 Visitor structure

Table 5 Class description of visitor

Class Name	Reminder
Descriptions	The client of the visitor pattern. This class will use the visitor to visit all the products and trading of a given user.

Class Name	NodeVisitor	
Descriptions	The abstract class of the visitor, it can visit class: Façade, Trading, Product. The real work that need to be done will be implemented in the concrete visitor classes.	
Abstract Function	Name	visitFacade
	Des	
Abstract Function	Name	visitTrading
	Des	
Abstract Function	Name	visitProduct
	Des	

Class Name	NodeVisitor	
Descriptions		
Attribute	Name	m_Reminder
	Des	
Function	Name	visitFacade
	Des	When visiting Façade, it will in turn visit each product in the Façade.productList
Function	Name	visitProduct
	Des	When visiting a product, it will in turn visit each trading in this product.
Function	Name	visitTrading
	Des	When visiting a trading, it will compare the current date and the due date of the trading and show the proper reminding information on the Reminder. (The client)

Class Diagram

Now we integrate all the patterns mentioned in the previous sections and all other classes that are used in the implementation of PTBS system to make the Class Diagram for the entire system. Figure 6 shows the Class Diagram for the PTBS system. All the patterns are highlighted with boxes.

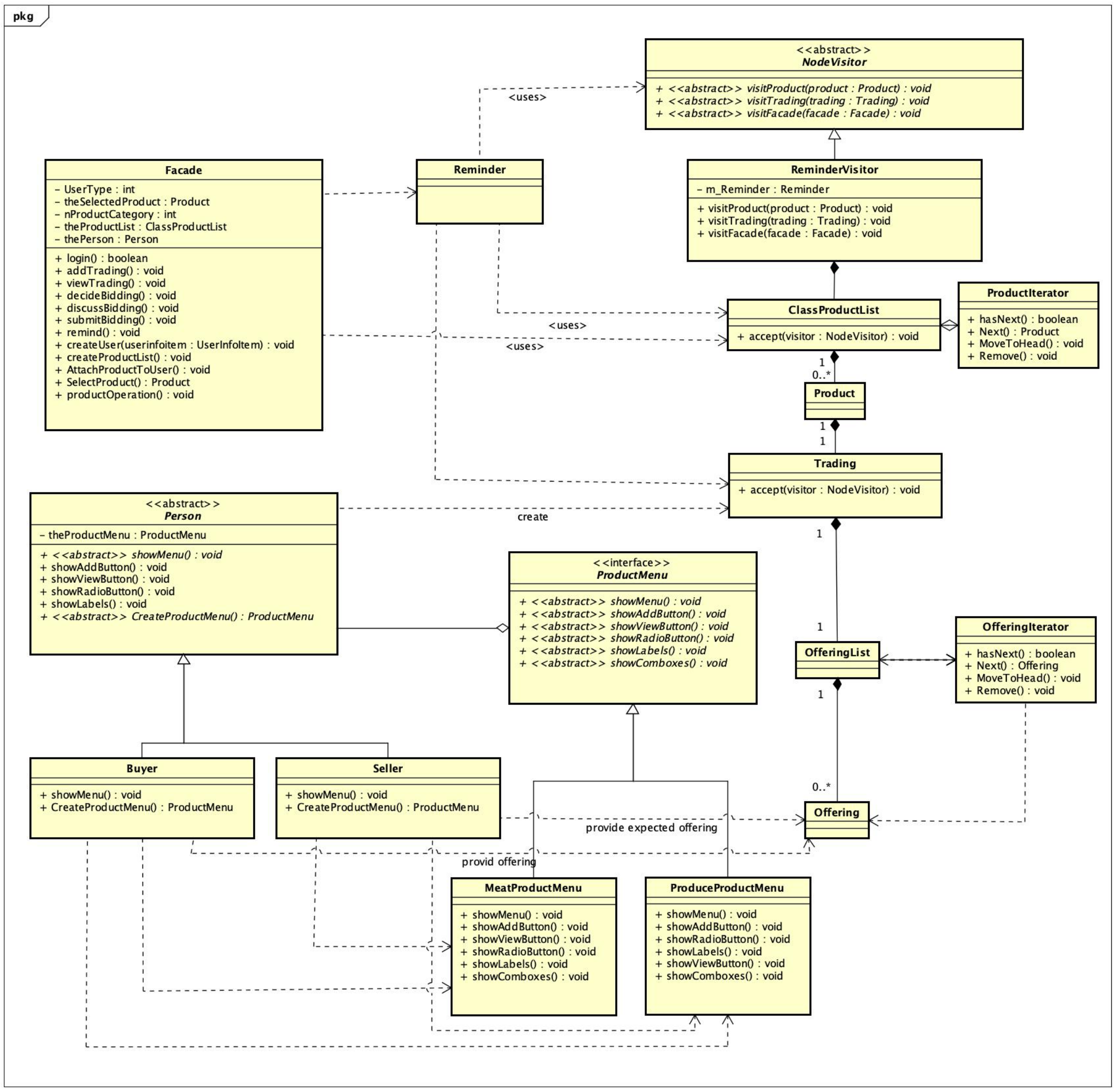
pkg
<<abstract>> NodeVisitor
+ <<abstract>> visitProduct(product : Product) : void
+ <<abstract>> visitTrading(trading : Trading) : void
+ <<abstract>> visitFacade(facade : Facade) : void
<uses>
Reminder
ReminderVisitor
- m_Reminder : Reminder
+ visitProduct(product : Product) : void
+ visitTrading(trading : Trading) : void
+ visitFacade(facade : Facade) : void
Facade
- UserType : int
- theSelectedProduct : Product
- nProductCategory : int
- theProductList : ClassProductList
- thePerson : Person
+ login() : boolean
+ addTrading() : void
+ viewTrading() : void
+ decideBidding() : void
+ discussBidding() : void
+ submitBidding() : void
+ remind() : void
+ createUser(userinfoitem : UserInfoItem) : void
+ createProductList() : void
+ AttachProductToUser() : void
+ SelectProduct() : Product
+ productOperation() : void
ClassProductList
+ accept(visitor : NodeVisitor) : void
<uses>
<uses>
ProductIterator
+ hasNext() : boolean
+ Next() : Product
+ MoveToHead() : void
+ Remove() : void
1
0..*
Product
1
1
Trading
+ accept(visitor : NodeVisitor) : void
<<abstract>> Person
- theProductMenu : ProductMenu
+ <<abstract>> showMenu() : void
+ showAddButton() : void
+ showViewButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ <<abstract>> CreateProductMenu() : ProductMenu
create
<<interface>> ProductMenu
+ <<abstract>> showMenu() : void
+ <<abstract>> showAddButton() : void
+ <<abstract>> showViewButton() : void
+ <<abstract>> showRadioButton() : void
+ <<abstract>> showLabels() : void
+ <<abstract>> showComboxes() : void
1
1
OfferingList
OfferingIterator
+ hasNext() : boolean
+ Next() : Offering
+ MoveToHead() : void
+ Remove() : void
1
0..*
Offering
Buyer
+ showMenu() : void
+ CreateProductMenu() : ProductMenu
Seller
+ showMenu() : void
+ CreateProductMenu() : ProductMenu
provide expected offering
provid offering
MeatProductMenu
+ showMenu() : void
+ showAddButton() : void
+ showViewButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ showComboxes() : void
ProduceProductMenu
+ showMenu() : void
+ showAddButton() : void
+ showRadioButton() : void
+ showLabels() : void
+ showViewButton() : void
+ showComboxes() : void

Figure 6 Class diagram of PTBS system

Additional information

Create a "database" with the following data:

Buyers:

Username	Password
tutu	1111
mimi	2222

Sellers:

Username	Password
pepe	3333

This will be the test set for your data...

Refer to the SellInfo.txt, BuyInfo.txt, ProductInfo.txt, UserProduct.txt for detailed info.

Submission

A zip file assignDP.<asurite>.zip (asurite should be replaced by your asurite, for me it would be assignDP.mjfindle.zip). Make sure you commented your code well and marked where you see yourDesign Patterns.