Assignment - Design Pattern

Version: October 4, 2019

Due: Friday Oct 17th, 11:59pm

Objectives

Use design patterns to build the application

General (Note the use of REQUIREMENTS 😊)

1. You MUST to create 5 design patterns for the application specified below (each worth 8 points).
2. You MUST use Java!
3. You MAY use Astah to generate the skeleton Java code from the UML. 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) have to have a running/working application (but it should not have compiler errors or warnings!).
5. We will grade you based on the code that you've 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 you can choose any 4 design patterns for task 1 that you see 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 relatively coding intensive again, start early it will take a while to figure things out.**

Introduce Design Patterns (40)

In this homework we implemented five design patterns into the Homework Assignment Distribution and Collection System (HACS). The five design patterns, which are implemented within the HACS systems, are **Façade, Bridge, FactoryMethod, Iterator, and Visitor**. In the following sections, a brief description of the patterns and the detail implementation of the patterns are presented.

1. Façade

The façade pattern can make the task of accessing a large number of modules much simpler by providing an additional interface layer. In the implementation of the HACS system, the façade lies in 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 HACS system. For example, in the HACS 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.

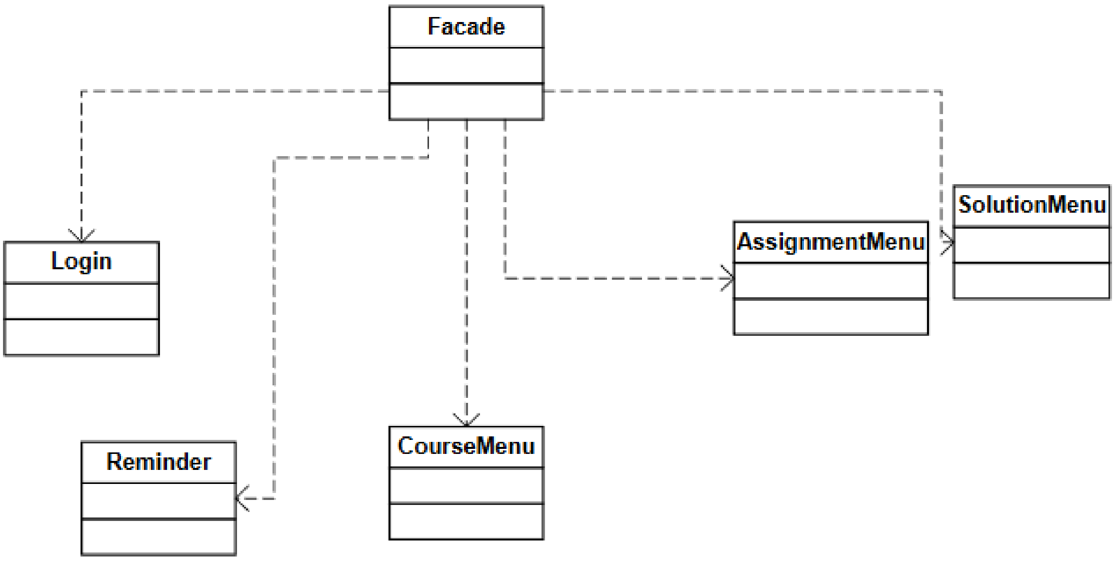
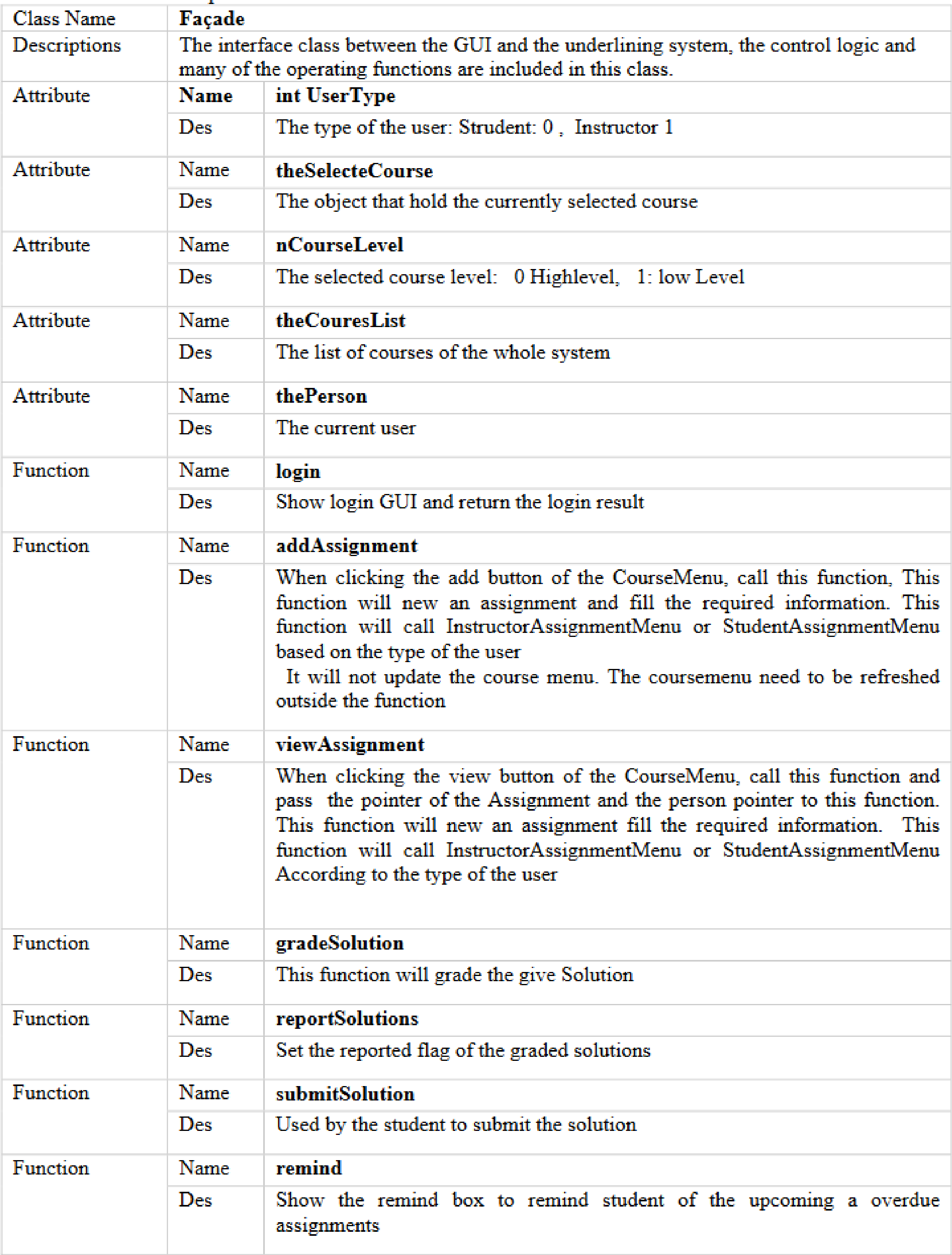
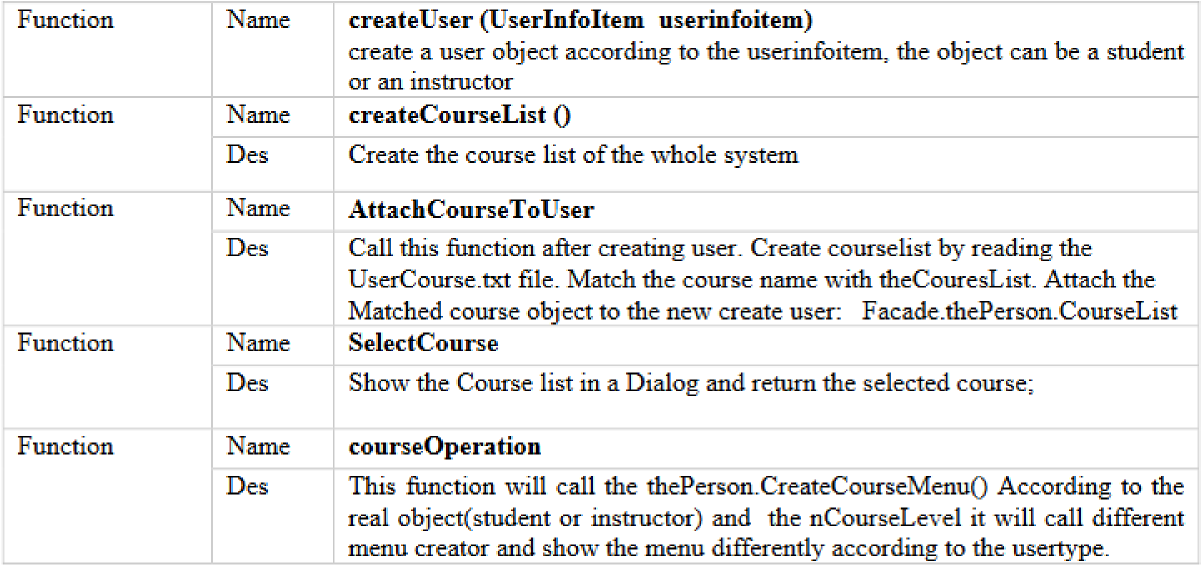


Figure 1 Façade Structure

Table 1 Class Description of Facade



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**	**int UserType**
	Des	The type of the user: Strudent: 0 , Instructor 1
Attribute	Name	**theSelecteCourse**
	Des	The object that hold the currently selected course
Attribute	Name	**nCourseLevel**
	Des	The selected course level: 0 Highlevel, 1: low Level
Attribute	Name	**theCouresList**
	Des	The list of courses of the whole system
Attribute	Name	**thePerson**
	Des	The current user
Function	Name	**login**
	Des	Show login GUI and return the login result
Function	Name	**addAssignment**
	Des	When clicking the add button of the CourseMenu, call this function, This function will new an assignment and fill the required information. This function will call InstructorAssignmentMenu or StudentAssignmentMenu based on the type of the user It will not update the course menu. The coursemenu need to be refreshed outside the function
Function	Name	**viewAssignment**
	Des	When clicking the view button of the CourseMenu, call this function and pass the pointer of the Assignment and the person pointer to this function. This function will new an assignment fill the required information. This function will call InstructorAssignmentMenu or StudentAssignmentMenu According to the type of the user
Function	Name	**gradeSolution**
	Des	This function will grade the give Solution
Function	Name	**reportSolutions**
	Des	Set the reported flag of the graded solutions
Function	Name	**submitSolution**
	Des	Used by the student to submit the solution
Function	Name	**remind**
	Des	Show the remind box to remind student of the upcoming a overdue assignments



Function	Name	**createUser (UserInfoItem userinfoitem)** create a user object according to the userinfoitem, the object can be a student or an instructor
Function	Name	**createCourseList ()**
	Des	Create the course list of the whole system
Function	Name	**AttachCourseToUser**
	Des	Call this function after creating user. Create courselist by reading the UserCourse.txt file. Match the course name with theCouresList. Attach the Matched course object to the new create user: Facade.thePerson.CourseList
Function	Name	**SelectCourse**
	Des	Show the Course list in a Dialog and return the selected course;
Function	Name	**courseOperation**
	Des	This function will call the thePerson.CreateCourseMenu() According to the real object(student or instructor) and the nCourseLevel 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 HACS system. When a user logs in, the bridge will help to load the appropriate menu for either student or instructor. Furthermore, the menu should be different depends on which kind of course is selected, i.e., high-level course or low-level course. This feature is implemented by bridge pattern.

In the bridge that is implemented in HACS, the CourseMenu is the implementor class hierarchy, which has two subclasses named HighLevelCourseMenu and LowLevelCourseMenu, and the Person is the abstraction class hierarchy which has two subclasses named Instructor and Student.

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 CourseMenu is configured at runtime depends on the type of the course (high level course or low-level course) and the type of the user (student or instructor). 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 course is added, we can extend the abstraction (person) or implementor (courseMenu) accordingly and independently. Without bridge, we would have to instantiate 4 subclasses for two kinds of user and two kinds of courses. Even worse, if we add one type of course, we would have to add two subclasses for both instructor and student. Figure 2 illustrates the structure of the Bridge pattern in this system. Table 2 shows the class description of Bridge pattern in this implementation.

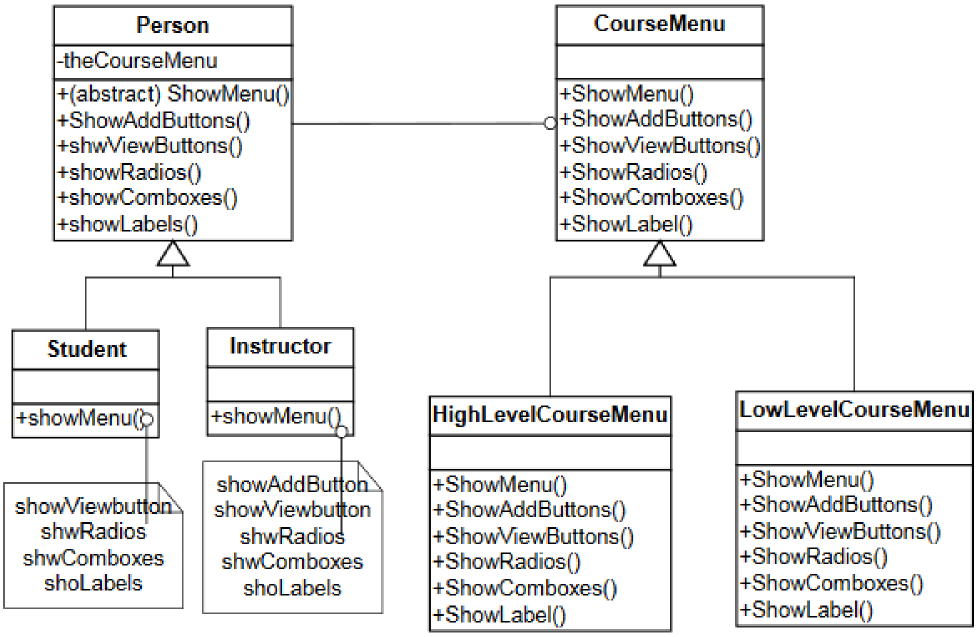


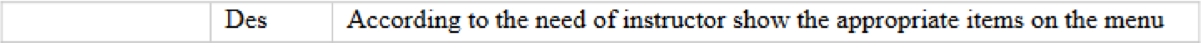
Figure 2 Bridge Structure

Table 2 Class Description of Bridge

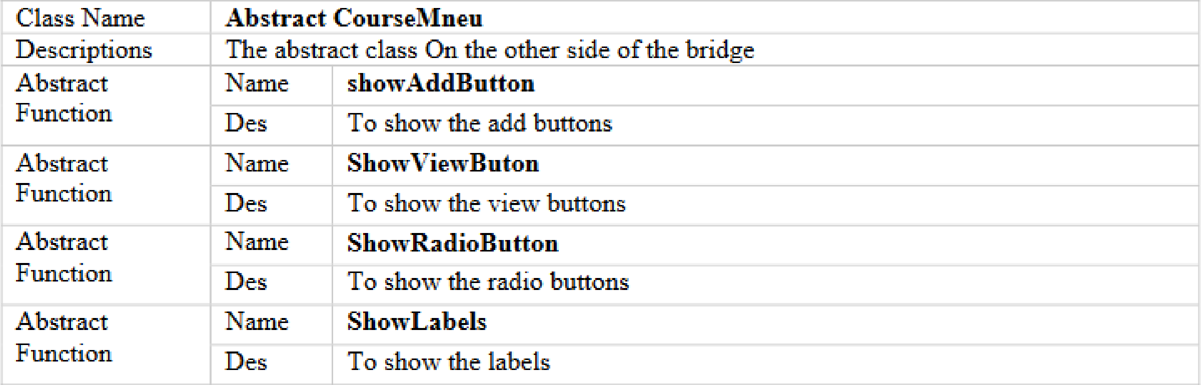
Class Name	**Abstract Person**	
Descriptions	The abstract class On one side of the bridge	
Attribute	Name	**theCourseMenu**
	Des	Variable of CourseMenu. Use this variable to point to a concrete courseMenu object. Later, it will operate the object.
Function	Name	**showAddButton**
	Des	Call the implementation to show the "add" buttons
Function	Name	**ShowViewButon**
	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	**Show**
	Des	Overrided by the class: student and instructor to show the menu

Class Name	Student	
Descriptions	The concrete subclass of Person	
Function	Name	**ShowMenu**
	Des	According to the need of student show the appropriate items on the menu

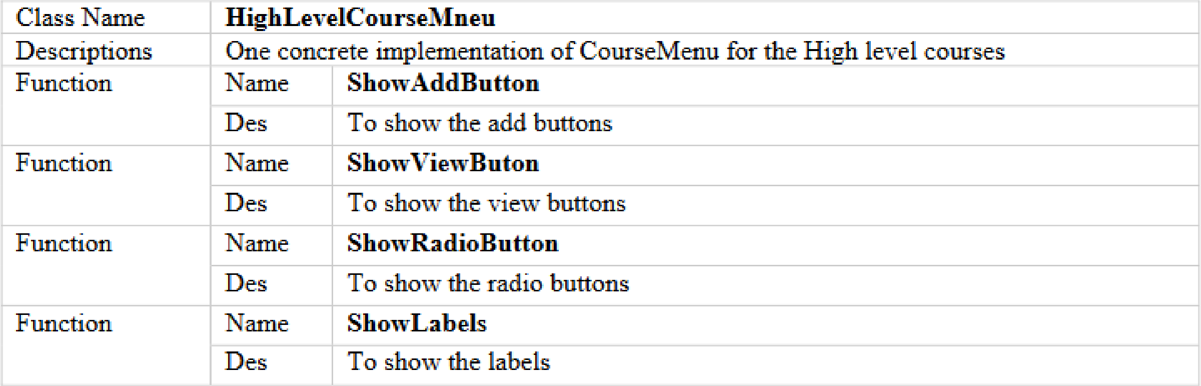
Class Name	**Instructor**	
Descriptions	The concrete subclass of Person	
Function	Name	**showMenu**



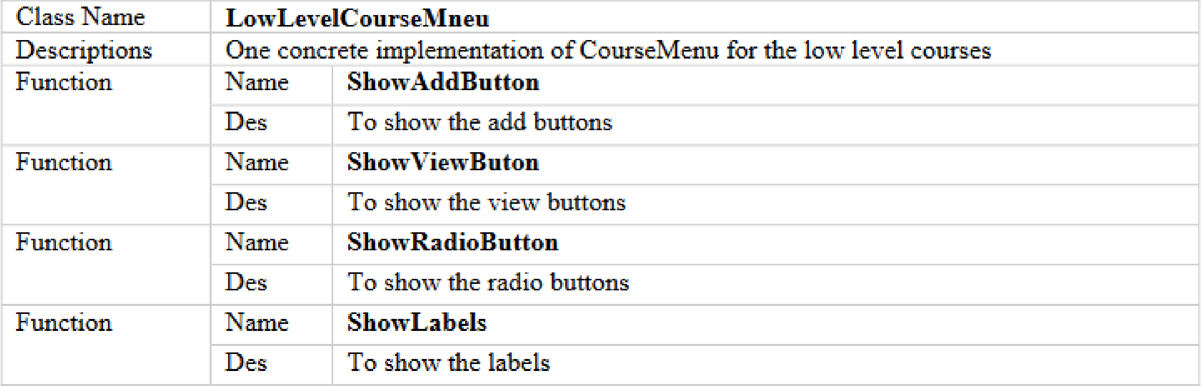
	Des	According to the need of instructor show the appropriate items on the menu



Class Name	**Abstract CourseMneu**	
Descriptions	The abstract class On the other side of the bridge	
Abstract Function	Name	**showAddButton**
	Des	To show the add buttons
Abstract Function	Name	**ShowViewButon**
	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	**HighLevelCourseMneu**	
Descriptions	One concrete implementation of CourseMenu for the High level courses	
Function	Name	**ShowAddButton**
	Des	To show the add buttons
Function	Name	**ShowViewButon**
	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	**LowLevelCourseMneu**	
Descriptions	One concrete implementation of CourseMenu for the low level courses	
Function	Name	**ShowAddButton**
	Des	To show the add buttons
Function	Name	**ShowViewButon**
	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 HACS system, the Factory Method pattern is implemented when the CourseMenu is loaded. The CourseMenu depends on course level and user type. The Factory Method will determine which class to instantiate.

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

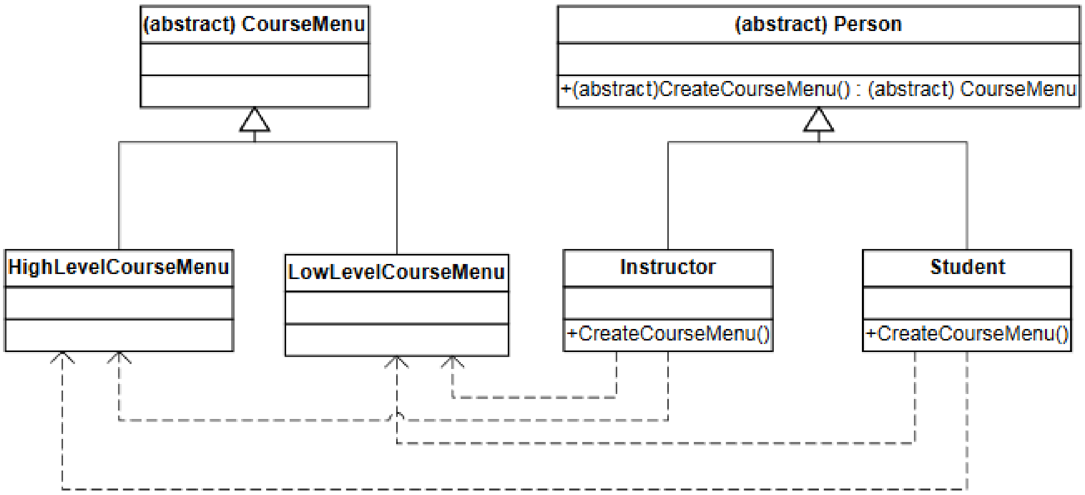


Figure 3 Factory Method Structure

Table 3 Class Description of Factory Method

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

Class Name	**Instructor**	
Descriptions	The concrete implementation of the Person class	
Function	Name	**CreateCourseMenu()**
	Des	According to the Course type create a concrete course menu: High level or Low level

Class Name	**Student**	
Descriptions	The concrete implementation of the Person class	
Function	Name	**CreateCourseMenu()**
	Des	According to the Course type create a concrete course menu: High level or Low level

Class Name	**Abstract CourseMenu**
Descriptions	The abstract product of the factor method

Class Name	**HighLevelCourseMenu**
Descriptions	A subclass of CourseMenu. One of the concrete product of the factor method

Class Name	**HighLevelCourseMenu**
Descriptions	A subclass of CourseMenu. One of the concrete product of the factor method

Iterator

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

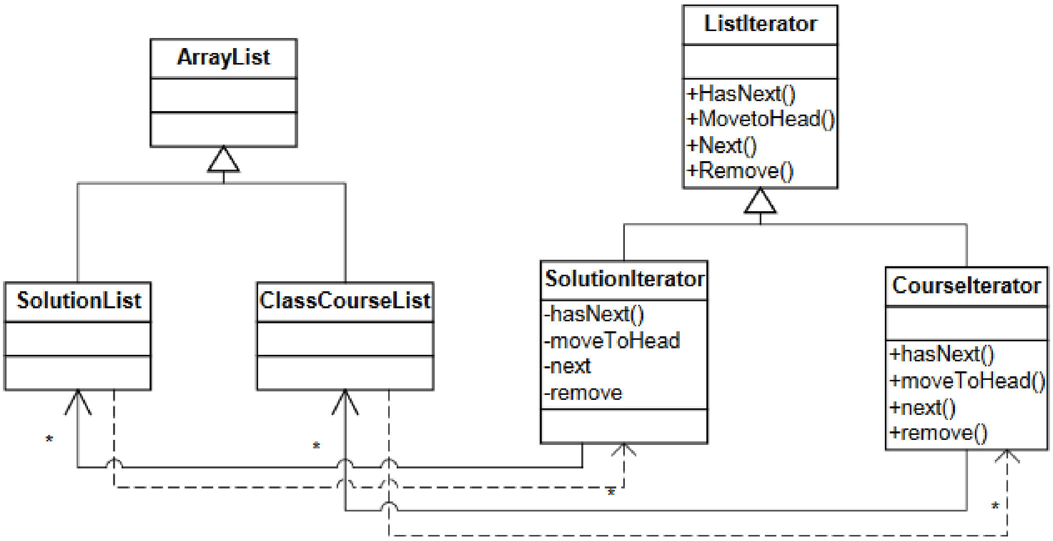


Figure 4 Iterator Structure

Table 4 Class Description of Iterator

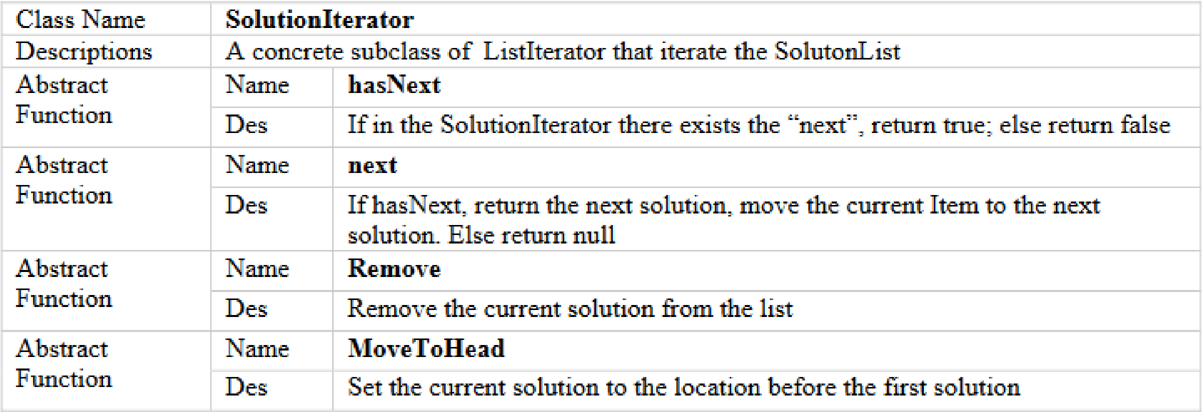
Class Name	**ArrayList**
Descriptions	The abstract class of the list to be iterated

Class Name	**SolutionList**
Descriptions	Subclass of ArrayList. One concreted List class that need to be iterated

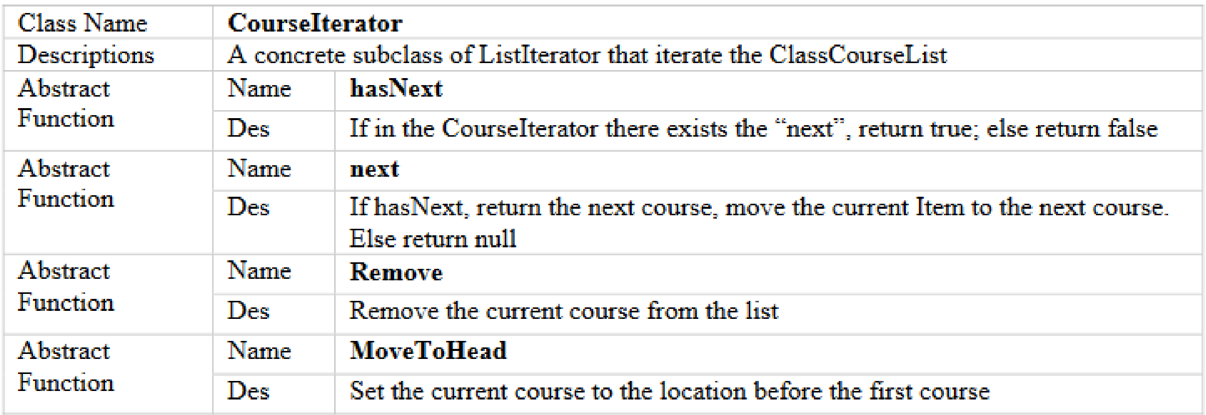
Class Name	**ClassCourseList**
Descriptions	The abstract class of the list to be iterated

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	**SolutionIterator**	
Descriptions	A concrete subclass of ListIterator that iterate the SolutonList	
Abstract Function	Name	**hasNext**
	Des	If in the SolutionIterator there exists the "next", return true; else return false
Abstract Function	Name	**next**
	Des	If hasNext, return the next solution, move the current Item to the next solution. Else return null
Abstract Function	Name	**Remove**
	Des	Remove the current solution from the list
Abstract Function	Name	**MoveToHead**
	Des	Set the current solution to the location before the first solution



Class Name	**CourseIterator**	
Descriptions	A concrete subclass of ListIterator that iterate the ClassCourseList	
Abstract Function	Name	**hasNext**
	Des	If in the CourseIterator there exists the "next", return true; else return false
Abstract Function	Name	**next**
	Des	If hasNext, return the next course, move the current Item to the next course. Else return null
Abstract Function	Name	**Remove**
	Des	Remove the current course from the list
Abstract Function	Name	**MoveToHead**
	Des	Set the current course to the location before the first course

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 of changing 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 HACS system, ReminderVisitor provides the Visitor capacity. Figure 5 illustrates the structure of Visitor pattern in HACS system. Table 5 lists the class description of Visitor Pattern.

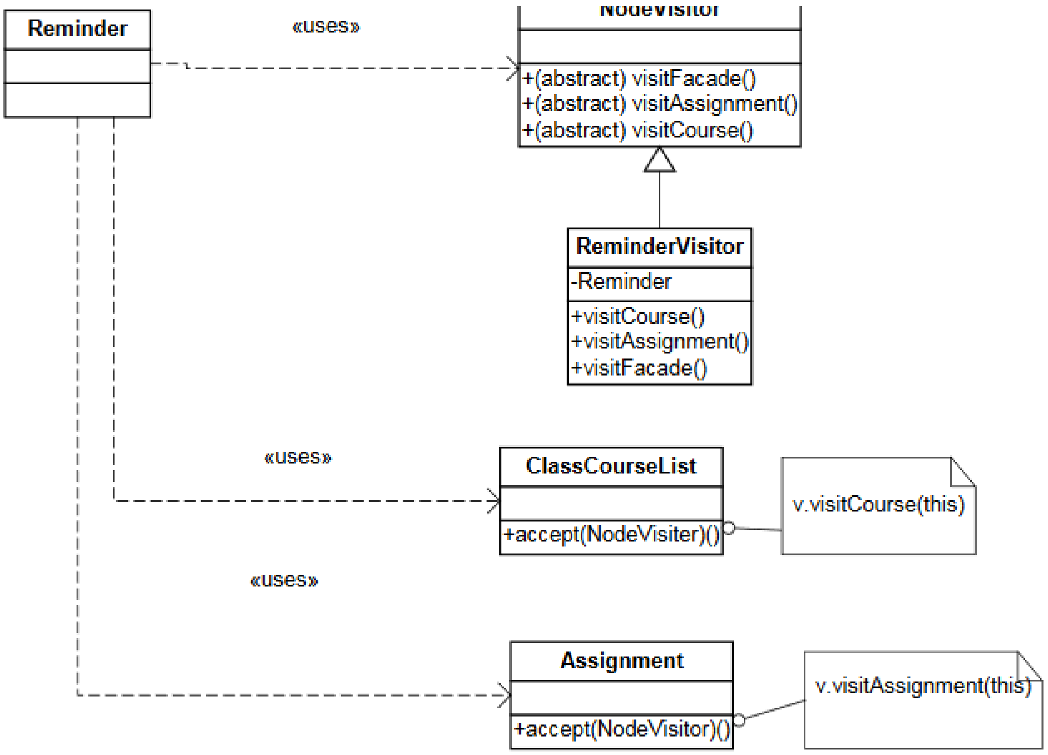


Figure 5 Visitor Structure

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 courses and assignments of a given user

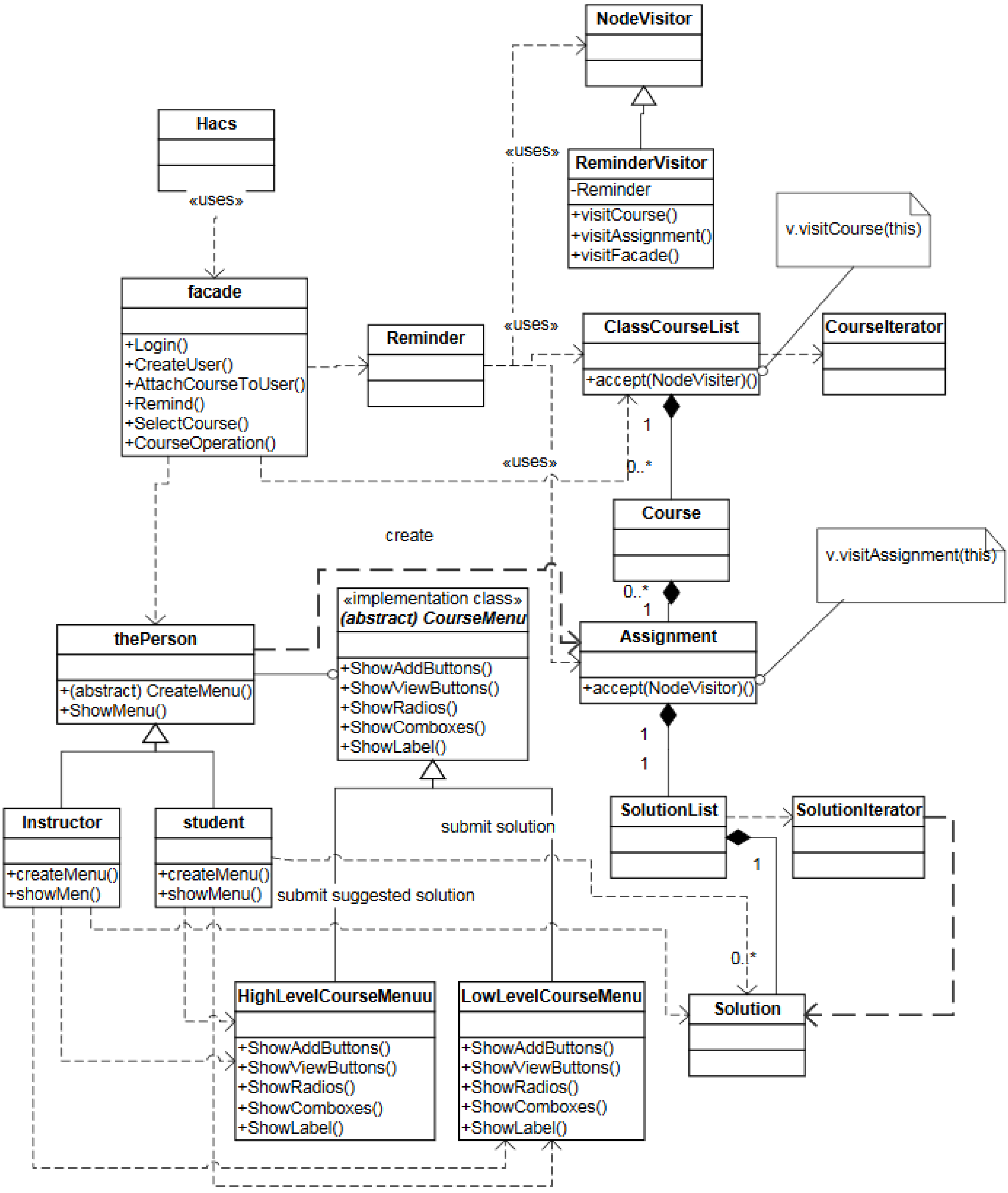
Class Name	**NodeVisitor**	
Descriptions	The abstract class of the visitor, it can visit class: Façade, Assignment, Course. The real work that need to be done will be implemented in the concrete visitor classes	
Abstract Function	Name	**visitFacade**
Abstract Function	Name	**visitAssignment**
Abstract Function	Name	**visitCourse**
	Des	

Class Name	**ReminderVisitor**	
Attribute	Name	**m_Reminder**

Function	Des	The client. The Reminder will call this
Function	Name	**visitFacade**
	Des	When visiting Façade it will in turn visit each course in the Façade.courselist
Function	Name	**VisitCourse**
	Des	When visiting a course, it will in turn visit each assignment in this course.
Function	Name	**visitAssingment**
	Des	When visiting an assignment, it will compare the current date and the due date of the assignment 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 HACS system to make the Class Diagram for the whole system. Figure 6 shows the Class Diagram for the HACS system. All the patterns are highlighted with boxes.

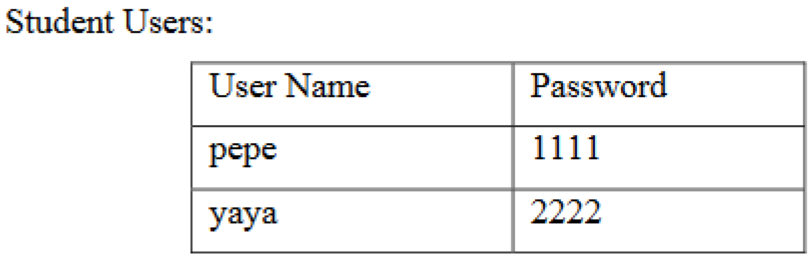


HACS Class Diagram

Figure 6 Class Diagram of HACS System

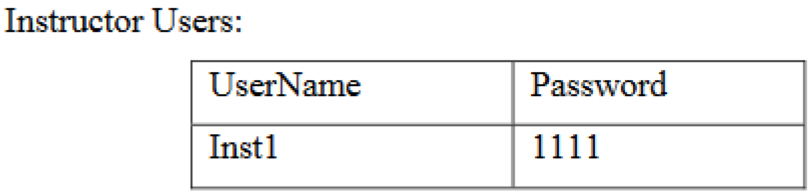
Additional information

Create a “database” with the following data:



Student Users:

User Name	Password
pepe	1111
yaya	2222



Instructor Users:

UserName	Password
Inst1	1111

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

Refer to the InsInfor.txt, StuInfo.txt, CourseInfo.txt, UserCourse.txt for detail information.

Submission

You need to submit

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