CS 6359 Homework #3

Student Name and Id: Naveenraj Palanisamy, nxp154130

Class and Section: CS 6359.002

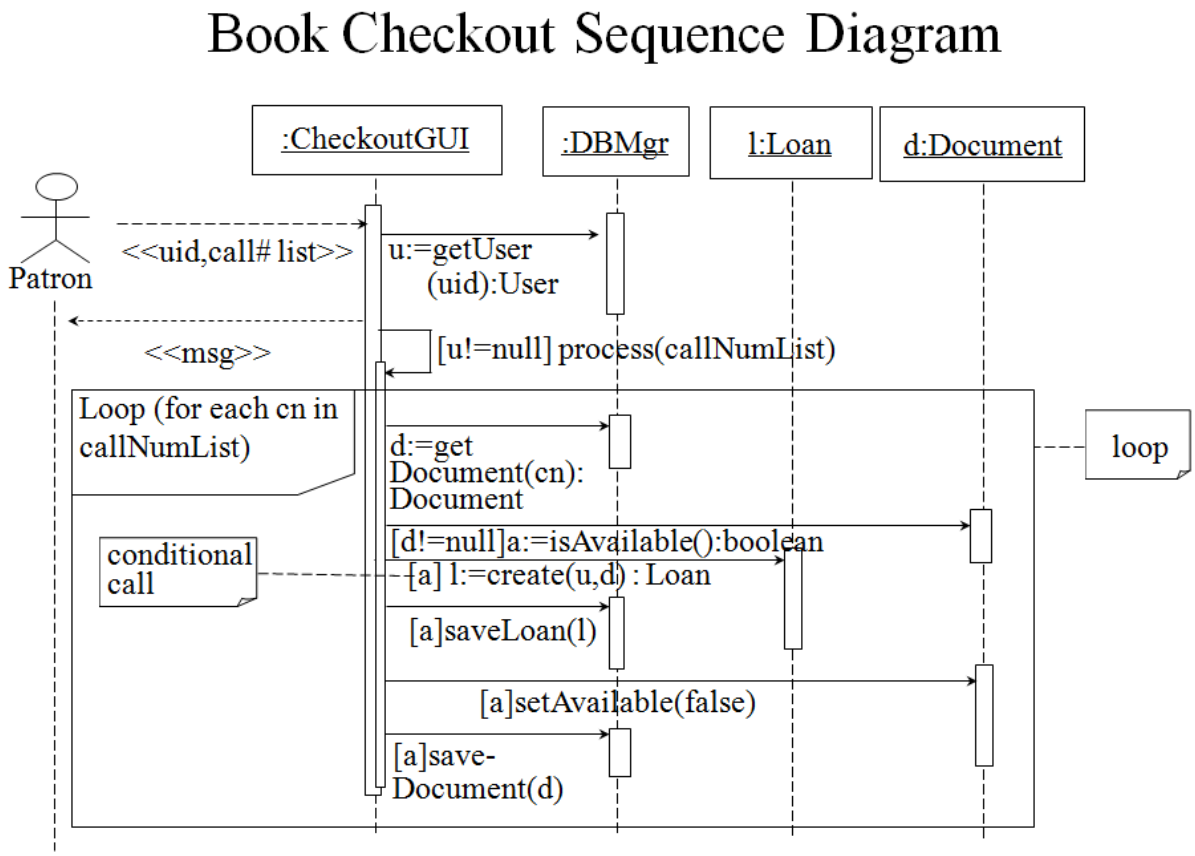


Figure 1: Sequence diagram of a use case: a user checkouts book(s) from a library

Question 1: [25]

The Figure 1 shows the sequence diagram of a use case where a user (also known as 'Patron') checks out book(s) from the library management system. From the sequence diagram, write down the implementation of 'process(callNumberList) method. Please note, the method call text is below the arrow-line.

BOOK_Store.zip

Attached full code document zip:

```
import DBcontroller.DBmgr;
import model.Document;
import model.Loan;
import model.User;

/**
 * Servlet implementation class checkoutGUI
 */
@WebServlet("/checkoutGUI")
public class checkoutGUI extends HttpServlet {
	DBmgr db;
	User u;

	checkoutGUI()
	{
		db=new DBmgr();
	}


	protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
		// TODO Auto-generated method stub
		u=db.getUser(request.getParameter("userid"));
		if(u!=null){
			process(request.getParameter("callist").toCharArray());
		}

	}
	public void process(char[] callist){

		Document d;
		Loan l;
		boolean avability;
		for(int i=0;i<callist.length;i++){
			d=db.getDocument(callist[i]);
			if(d!=null)
			{
				avability=d.isAvailable();
				if(avability){
					l=new Loan(u,d);
					db.saveLoan(l);
					d.setAvailable(false);
					db.saveDocument(d);
				}
			}
		}
	}

}
```

Question 2: [15]

This is a problem about the GRASP design patterns. How does the Creator pattern lower coupling in a design? Describe very briefly.

Creator pattern is that which class is responsible of creating object of another class. Take class B should be responsible to create Object of Class A. If B instance completely aggregates instance of A, B object records A object. B closely uses A object. B has initialization parameter of A.

Example of Creator Pattern: Sale creates instance of SaleLineItem. Since Sale has all the initialization parameter of SaleLineItem, here Sale will act as the creator of SaleLineItem.

Creator Helps to achieve low coupling in the design: Since Class B creates Object of A, there will be tight coupling between the Class B and A. This pattern will helps to **achieve low coupling in the overall system**, since it prevents any other coupling to the class A.

Example of Creator achieves low coupling in the overall system: Since Sale is the creator of SaleLineItem, SaleLineItem will be tightly coupled only to sale. In this case it will prevent any other coupling to SaleLineItem. Now by the expert pattern since Sale has the information of SaleLineItem, Sale will be contacted directly instead of going to both Sale and SaleLineItem, It maintains low coupling in the System.

Question 3: [15]

This is another problem about GRASP design patterns. Consider a design that we used an interface, InterfaceA (with an methodA() method), and several classes that implemented it such as: ClassB, ClassC, ClassD, ClassE, ClassF, and ClassG. In terms of cohesion, what would be the disadvantages of combining all of these into one class, ClassAll?

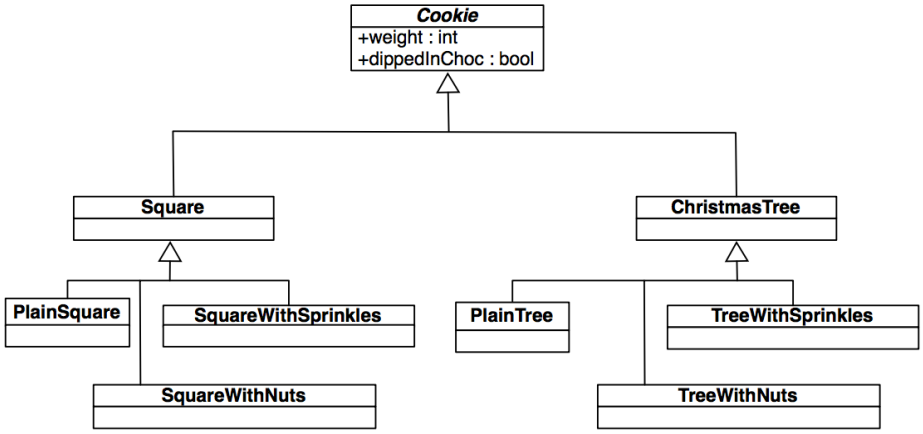
When all the classes are combined to the one class then cohesion of the class will be reduced. Class **ClassAll will be now with low cohesion**, because of doing all the functionalities of the classes (ClassB, ClassC, ClassD, ClassE, ClassF, and ClassG) and also it will make ClassAll Difficult to maintain. **Even High coupling can be permitted in some case but low cohesion class design is not at all accepted**.

Example: Take Animal as the interface and it has method as makesound(), now we have various classes like Dog,Cat and Lion implementing Animal. Each Animal will make different sound. Now if we combain all in the one class as AnimalAll, it has to preform itself all the sounds of the animal. It will make AnimalAll low cohesion. And also it will make AnimalAll difficult to maintain.

Question 4: [45]

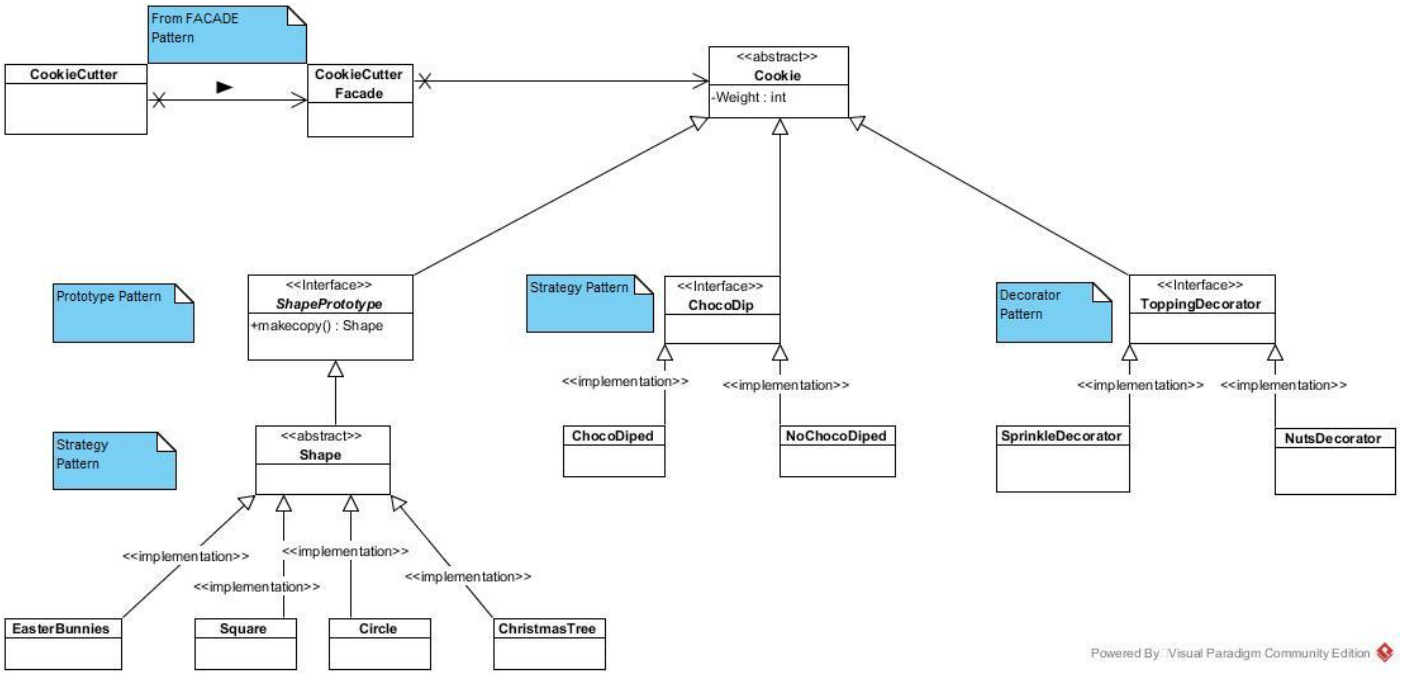
You have been asked to design a new version of CookieCutterPro, the latest version of a software package to automate the production of a variety of baked goods.

CookieCutterPro can produce cookies of various shapes: simple squares and circles, but also seasonal shapes, like christmas trees and easter bunnies. Cookies may be decorated with sprinkles or nuts. Some cookies are dipped in chocolate. Every cookie has a weight in grams. After this discussion, a colleague proposes the following design:



A) This design favor inheritance over aggregation. Give an alternative design in the form of an UML diagram. Explicitly name any design patterns or principles you choose to apply. (strategy to choose nuts or sprinkles, prototype to create clone objects, decorator to create toppings, façade to simplify the code by having extra class to handle everything)

Created UML diagram for the cookie cutter. Cookie cutter will give instruction to **façade** based on the instruction façade will choose operations to be performed. Façade is used here because client doesn't want to know exact implementations and client will be overloaded. **Strategy** pattern is applied for the chocolate dips selection and shape selection. **Prototype** pattern is used for the shapes because we need not create object each time we can reuse the same shape created last time. **Decorations** pattern is applied for the toppings this will make dynamic update of toppings.



B) These cookies are produced by different machine. Every machine can make any kind of cookie, but requires slightly different instructions to do so. All the machines support the same operations: addTopping(), dipInChocolate(), cutShape(), and bake(). Revise your design in form of UML diagram to take this requirement into account by using "bridge" design patterns.
Note: Please first describe (very briefly) the **bridge design pattern**. What is the problem, what is the solution. Then apply it to this problem.

Bridge design pattern is decoupling of abstraction from implementation using encapsulation, aggregation and inheritance. Bridge design pattern can be used when we are able to decouple abstract class from concreate class and also when we want abstract class to define some rules and concreate class to add more rules on it.

Problem: Now we have several machine each do all the operations but with slightly different instruction. So now we need all the machine to have access to access for all the cookie types.

Solution: created Bridge between the machines and the abstract cookies. There **is aggregation between the implementation class that is machine and the abstract class cookie**. Now each machine will have access to all the cookie types and can produce cookies of all shapes with decorations and chocolate dips with different instructions.