Student ID_____**Student Name**_____

Advanced Software Development DE

Midterm Exam August 17 2019

PRIVATE AND CONFIDENTIAL

1. **Allotted exam duration is 2 hours.**
2. **Closed book/notes.**
3. **No personal items including electronic devices (cell phones, computers, calculators, PDAs).**
4. **Cell phones must be turned in to your proctor before beginning exam.**
5. **No additional papers are allowed. Sufficient blank paper is included in the exam packet.**
6. **Exams are copyrighted and may not be copied or transferred.**
7. **Restroom and other personal breaks are not permitted.**
8. **Total exam including questions and scratch paper must be returned to the proctor.**

6 blank pages are provided for writing the solutions and/or scratch paper. All 6 pages must be handed in with the exam

BE VERY CAREFUL WITH THE GIVEN 2 HOURS AND USE YOUR TIME WISELY. THE ALLOTED TIME IS GIVEN FOR EVERY QUESTION.

Write your name and student id at the top of this page.

Question 1 [40 points] {50 minutes}

Suppose you need to design and implement a shooting game. The game has the following requirements:

- The game has 3 levels: level 1, level 2 and level 3.
- When you have more than 10 points in level 1, you get an upgrade to level 2
- When you have more than 20 points in level 2, you get an upgrade to level 3
- If you get an upgrade from level 1 to level 2, you get 1 bonus point
- If you get an upgrade from level 2 to level 3, you get 2 bonus points
- Every time you hit the target in level 1, you get x points, where x = a random number between 0 and 6.
- Every time you hit the target in level 1, you get 2 * x points, where x = a random number between 0 and 6
- Every time you hit the target in level 1, you get 3 * x points, where x = a random number between 0 and 6
- When you start the game, you start with 0 points in level 1.

One of your colleagues wrote the following implementation:

```
public class GUI {
	public static void main(String[] args) {
		Game game = new Game();
		game.hit();
		game.hit();
		game.hit();
		game.hit();
		game.hit();
	}
}
```

```
public class Game {
  private int totalPoints = 0;
  private int level = 1;

  public void hit() {
     Random random = new Random();
     addPoints(random.nextInt(7));
     System.out.println("points="+totalPoints+"
                        level="+level);
  }

  public int addPoints(int newPoints) {
     if (level == 1) {
       totalPoints = totalPoints + newPoints;
       if (totalPoints > 10) { // move to level 2
         level = 2;
         totalPoints = totalPoints + 1;//add 1 bonus point
       }
     } else if (level == 2) {
       totalPoints = totalPoints + 2 * newPoints;
       if (totalPoints > 20) { // move to level 3
         level = 3;
         totalPoints = totalPoints + 2; //add 2 bonus
                                        //points
       }
     } else if (level == 3) {
       totalPoints = totalPoints + 3 * newPoints;
     }
     return totalPoints;
  }
}
```

If you run the application, then the output will look like this:

```
points=6 level=1
points=7 level=1
points=10 level=1
points=17 level=2
points=27 level=3
```

But now your team gets a new requirement that the given requirements about levels, points, bonus points and upgrades must be very flexible. The problem with the given code is that the addPoints() method contains a complex if-then-else structure that is OK for simple cases, but get complex and error prone when the business rules of the game get complex.

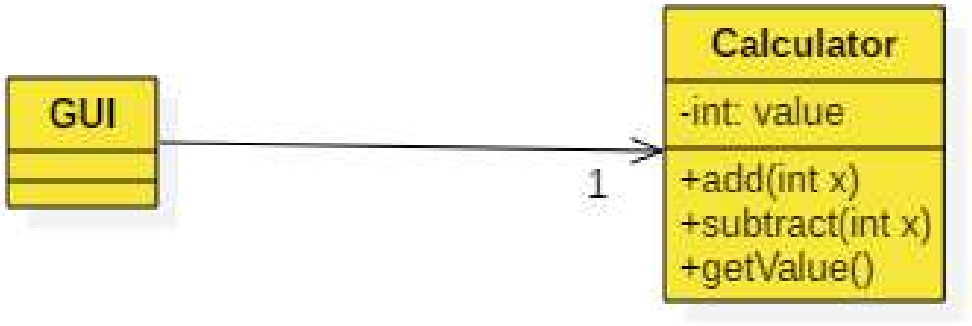
Rewrite the given code so that it is much easier to change the business rules of the game, even when they become complex. Do **NOT** change the given GUI class. (The answer should be given in Java code)

```
public class Game {
	private int totalPoints = 0;
	private ILevel level =  new Level1();

	public void hit() {
		Random random = new Random();
		addPoints(random.nextInt(7));
		System.out.println("points="+totalPoints+"
		  level="+level.getLevel());
	}

	public int getTotalPoints() {
		return totalPoints;
	}

	public void setLevel(ILevel level) {
		this.level = level;
	}

	public int addPoints(int newPoints) {
	    totalPoints = totalPoints + level.computePoints(this,
	                  newPoints);
		return totalPoints;

	}
}

public interface ILevel {
  int computePoints(Game game,int newPoints);
  String getLevel();
}


public class Level1 implements ILevel {

	public int computePoints(Game game, int newPoints) {
		int totalPoints = game.getTotalPoints();
		totalPoints = totalPoints + newPoints;
		if (totalPoints > 10) {
			game.setLevel(new Level2());
		}
		return totalPoints;
	}

	public String getLevel(){
		return "level 1";
	}
}
```

```
public class Level2 implements ILevel{


	public int computePoints(Game game,int newPoints) {
		int totalPoints = game.getTotalPoints();
		totalPoints = totalPoints + newPoints*2;
		if (totalPoints> 20){
			game.setLevel(new Level3());
		}
		return totalPoints;
	}

	public String getLevel(){
		return "level 2";
	}
}


public class Level3 implements ILevel{


	public int computePoints(Game game, int newPoints) {
		int totalPoints = game.getTotalPoints();
		totalPoints = totalPoints + newPoints*3;
		return totalPoints;
	}

	public String getLevel(){
		return "level 3";
	}
}
```

Question 2 [40 points] {40 minutes}

Suppose we have the following simple calculator application:

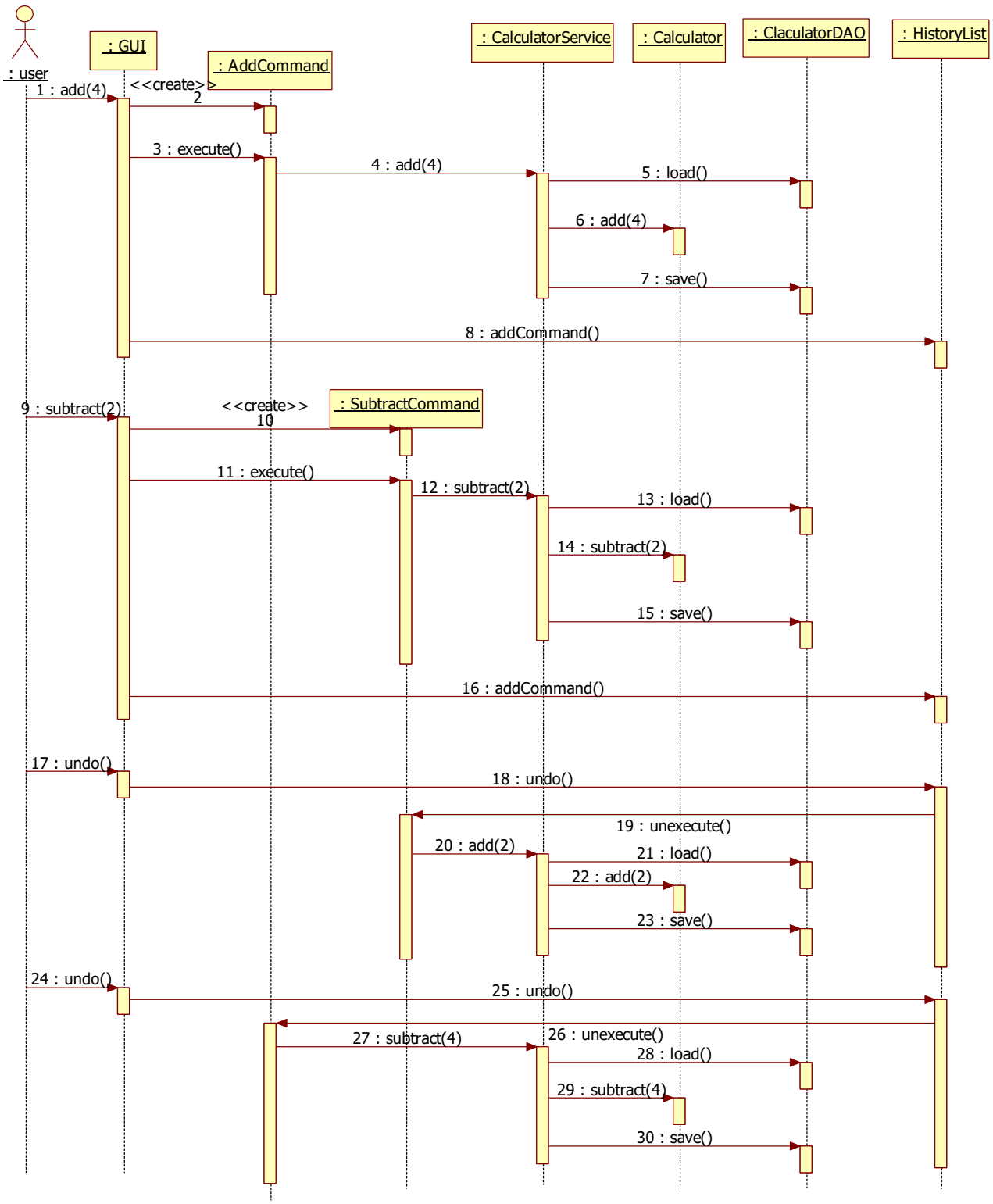


When we start the application, a new Calculator is created with value=0. Then we can perform the add and subtract operations on the calculator. The GUI will show the current value of the Calculator. Now we get 2 new requirements:

1. We need to support undo/redo functionality for the calculator
2. For every add() or subtract() action, we first need to retrieve the calculator value from the database, then do the add or subtract action on the calculator, and finally store the calculator value in the database. This way the database always has the latest value of the calculator.

It is your task to modify the design so that it supports the 2 new requirements.

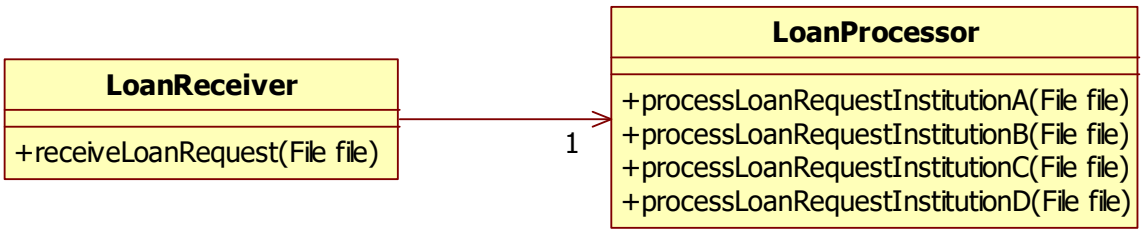
a. Draw the sequence diagram of the following scenario:
    1. We start the calculator application
    2. The user adds the value 4 to the calculator
    3. The user subtracts the value 2 from the calculator
    4. The user performs the undo action
    5. The user performs the undo action again
b. Write the complete Java code of one Command class. You can choose yourself which Command class.


: user
: GUI
: AddCommand
: CalculatorService
: Calculator
: ClaculatorDAO
: HistoryList
1 : add(4)
<<create>>
2
3 : execute()
4 : add(4)
5 : load()
6 : add(4)
7 : save()
8 : addCommand()
9 : subtract(2)
<<create>>
10
: SubtractCommand
11 : execute()
12 : subtract(2)
13 : load()
14 : subtract(2)
15 : save()
16 : addCommand()
17 : undo()
18 : undo()
19 : unexecute()
20 : add(2)
21 : load()
22 : add(2)
23 : save()
24 : undo()
25 : undo()
27 : subtract(4)
26 : unexecute()
28 : load()
29 : subtract(4)
30 : save()

```
public class AddCommand implements ICommand{
	private CalculatorService calculatorService;
	private int value;

	public AddCommand(CalculatorService calculatorService, int value) {
		this.calculatorService = calculatorService;
		this.value = value;
	}

	public void execute() {
		calculatorService.add(value);
	}

	public void unExecute() {
		calculatorService.subtract(value);
	}
}
```

Question 3 [15 points] {20 minutes}

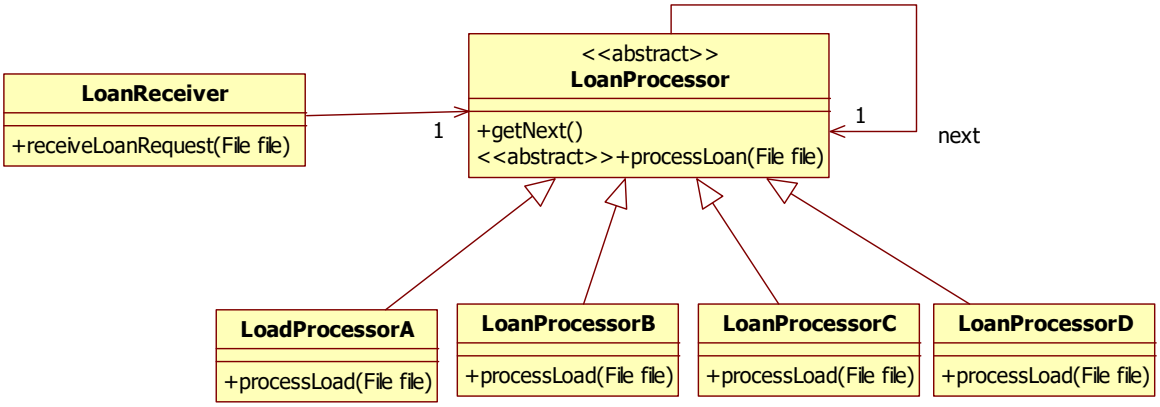
Suppose we have an existing loan application with the following design:

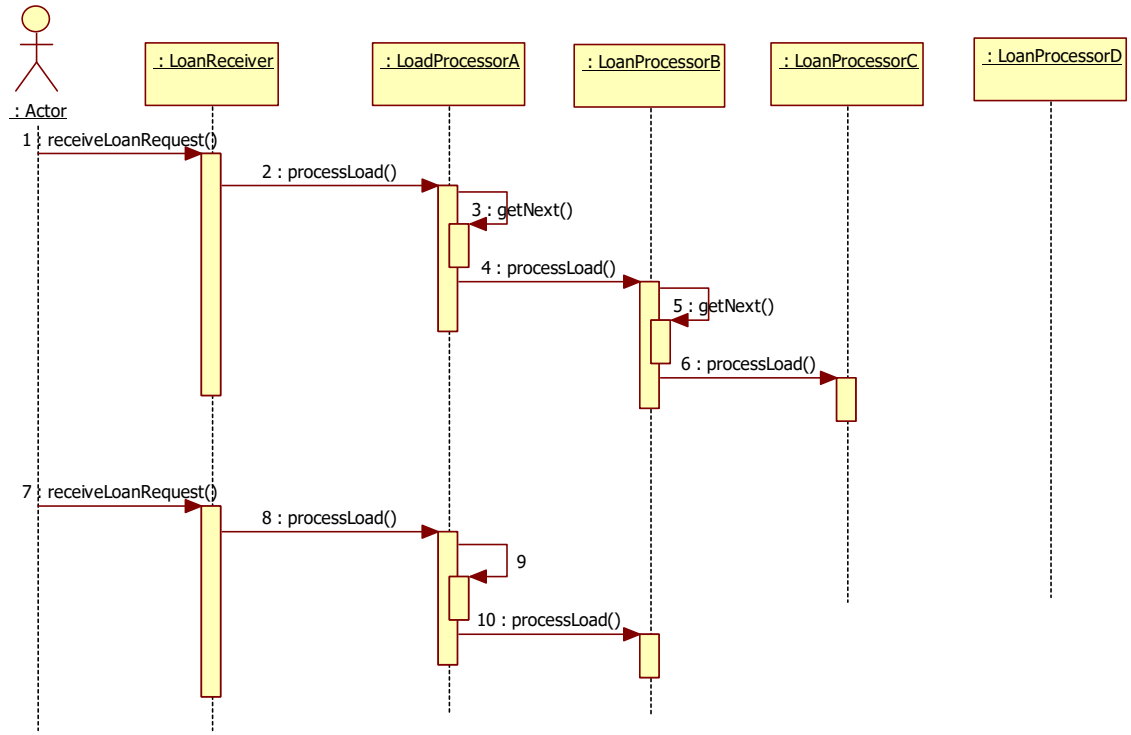


The loan application receives requests from different financial institutions, creates a loan offer and send this load offer back to the institution that submitted the request. The basic problem is that the loan offer request we receive from every institution is different. They all send a file, but they use different file formats, different data formats and the data is structured in a different way for every institution. The current application can handle only 4 institutions (A to D).

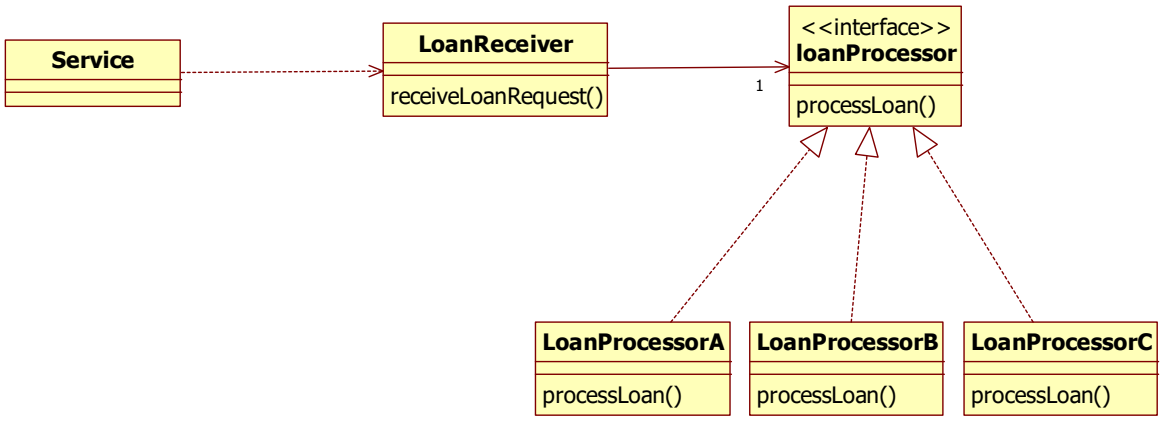
It is your job to create a better design than the existing application. Your design should support processing loan offers from many financial institutions and it should be easy to add support for processing loan offers from new financial institutions.

a) **Draw the class diagram of your design**
b) **Draw a sequence diagram that shows how your design works. Show the scenario of the following actions:**
    1. **Receive a loan request from institution D**
    2. **Receive a loan request from institution B**


<<abstract>>
LoanProcessor
+getNext()
<<abstract>>+processLoan(File file)
1
1
next
LoanReceiver
+receiveLoanRequest(File file)
LoadProcessorA
+processLoad(File file)
LoanProcessorB
+processLoad(File file)
LoanProcessorC
+processLoad(File file)
LoanProcessorD
+processLoad(File file)


: Actor
: LoanReceiver
: LoadProcessorA
: LoanProcessorB
: LoanProcessorC
: LoanProcessorD
1 : receiveLoanRequest()
2 : processLoad()
3 : getNext()
4 : processLoad()
5 : getNext()
6 : processLoad()
7 : receiveLoanRequest()
8 : processLoad()
9
10 : processLoad()

Strategy does not solve the problem



Question 4 [5 points] {10 minutes}

Describe how we can relate the **Observer** pattern to one or more principles of SCI. Your answer should be about half a page, but should not exceed one page (handwritten). The number of points you get for this questions depends how well you explain the relationship between the **Observer** pattern and one or more principles of SCI.