• You have 135 minutes, from 1:45pm – 4:00pm, to complete this exam. 
  You will lose credit if you keep working after the instructor calls for your papers. 
• This test is open-book/notes. You must work alone and may not use any computing devices of any kind. 
  If you violate the University Code of Conduct, you may receive a 0% for this exam and possibly further 
  punishment. 
• Write legibly and case-sensitively. **Do not abbreviate Java code.** You don't need to write `imports`. 
• You do not need to comment your code, except if the question says to do so. 
• Please be quiet during the exam. If you have a question or need, please raise your hand. 
• Corrections or clarifications to the exam will be written at the front of the room. 
• When you have finished the exam, please turn in your exam quietly and leave the room. 

Good luck! -- Marty

Score summary: (for grader use only)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
<th>Max</th>
<th>Earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Software Lifecycle</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Use case</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>UML Class Diagram</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>JUnit Test Case</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>Total Points</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. **Software Lifecycle**

According to what we learned in class, which of the following papers is most closely related to performing each of the following actions i -- xii? Choose from the following choices. Exactly one answer is correct for each question. Each choice might be used more than once. Write your answers on the right in the blanks provided.

(a) Boehm, "Spiral Model of Software Development"
(b) Faulk, "Software Requirements: A Tutorial"
(c) Cockburn, "Writing Effective Use Cases"
(d) Snyder, "Paper Prototyping"
(e) Beck, "A Laboratory for Teaching Object-Oriented Thinking"
(f) Riel, "Object-Oriented Design Heuristics"
(g) Fowler, "UML Distilled (Class Diagrams and Sequence Diagrams)"
(h) Horstmann, "Guidelines for Class Design"
(i) Sun Microsystems, "Java Coding Conventions"
(j) Cockburn/Williams, "Costs and Benefits of Pair Programming"
(k) Bruegge/Dutoit, "Testing"
(l) JUnit "Test Infected: Programmers Love Writing Tests"
(m) Geary, "Simply Singleton" / "Make Your Apps Fly" / "An Inside View of Observer"

i. **Our banking ATM system has an important scenario where the customer withdraws money. What should be the interaction steps and failure-handling extensions between the system and the customer?**

ii. **The ATM class is responsible for validating user input and create new accounts. It should talk to the SavingsAccount and CheckingAccount classes.**

iii. **We have created too many redundant User objects with the same internal state. How can we improve it?**

iv. **We have decided to use a portable Java graphical interface that will operate on Linux and Windows. Where should we record this decision?**

v. **The AccountSystem class has too much control over the system. What should we do about it?**

vi. **When I call deposit on an Account object, it must be true that the amount is non-negative, and at the end of the method, the account's balance may not be negative.**

vii. **CheckingAccount and SavingsAccount are both kinds of Accounts. Can we document this relationship?**

viii. **We're coding the account sorting algorithm using bubble sort. It might be too slow. When will we notice this?**

ix. **We need to talk about how the GUI would look when the user interacted with the various on-screen controls. What happens when he/she clicks the Create Account button?**

x. **Our developers don't want to do code reviews. Is there any other way we can integrate oversight into the development process?**

xi. **The data access subsystem has a fault that it sometimes erases a new account's data. We notice this when an "account not found" error appears on the screen. What's going on?**

xii. **We are going to store our bank account data in an Oracle database for speed. Should we write down that we've decided to do this?**
2. Use Case
You have been hired as a consultant to design the software for a Digital Video Recorder (DVR) product called NuVo. Consider the following specification of the NuVo software system:

NuVo is a hardware/software system that allows the user to digitally record video from a TV set to play back later. The NuVo device is hooked up to a TV and is controlled by a remote control. It can record a television show from a specified channel. After the show is recorded, it can be played back any number of times. The show can also be rewound and fast-forwarded.

The shows that are recorded by NuVo can be categorized into specific genres. There are also different kinds of shows: single episodes, pay-per-view shows, and series, which are comprised of a fixed number of episodes of the same show. A show is on a particular channel and has a definite start time and length and an MPAA rating.

NuVo can be utilized by several users. Each user can record his/her own show(s) as well as purchase pay-per-view shows. There are two types of users: general users have access to all channels and times, while children are limited based on the ratings of a show. Users have passwords that are used to gain access. Users have a set amount of time that they can record based on the plan that they have purchased. NuVo keeps track of the amount of record time left.

Based on this specification, write a formal use case for a "Record a TV Show" scenario for the system specified. Include all the applicable elements of formal use cases that we have discussed in class. You may use the next page if you need it.
2. Writing Space
3. UML Class Diagram

Draw a UML class diagram for the *NuVo* system specified on the last page. Your diagram should show all classes, methods, fields, access modifiers, and relationships between classes/objects with appropriate adornments in proper UML syntax. Follow appropriate object-oriented design guidelines as discussed in class. Remember that you do not need to write *get* and *set* methods for your classes’ fields.
4. Implementation

The following code implements a sorted collection of accounts for a banking system. Modify the AccountCollection class code below as specified. Write your answer on the next page. Specify your answer with statements such as, "At location (d), add the following statements: int x = 7; ...." See the Account class reference on the last page.

i. Use the Observer design pattern as specified in Inside Observer:
   - Make it possible for other classes to observe the state of an AccountCollection. Each time the state is changed in any way, tell the observers about it.

ii. Add code to test the following pre- and post-conditions as specified in Programming by Contract:
   - **Precondition:** At the start of the add method, account should be non-null. if it is null, an IllegalArgumentException should be thrown.
   - **Precondition:** At the start of all methods in AccountCollection, the accounts list should be non-null. if it is null, the author of AccountCollection has made a coding error.
   - **Postcondition:** Every account is supposed to be added to the list in sorted order by ID. This should remain true after a call to add or a call to load. If it is not the case, the author of AccountCollection has made a coding error.
   - **Postcondition:** At the end of the applyFees method, every account's balance should be non-negative. (The Account class does not check for this in its deposit and withdraw methods.) If an account has a negative balance after applyFees, the author of AccountCollection has made a coding error.

```java
import java.util.*;

/** A list of bank accounts. */
public class AccountCollection {
    private ArrayList accounts = new ArrayList();

    /** Add the given account to this collection, in sorted order. */
    public void add(Account account) {
        int index = 0; // find correct place to put account
        while (index < accounts.size()) {
            Account current = (Account)accounts.get(index);
            if (account.getID().compareTo(current.getID()) > 0)
                break;
            index++;
        }
        accounts.add(index, account);
    }

    /** Charge a $1.00 fee to every account. */
    public void applyFees() {
        for (Iterator i = iterator(); i.hasNext(); ) {
            Account account = (Account)i.next();
            account.withdraw(1.00);
        }
    }

    /** Reads a list of accounts from disk. */
    public void load() {
        accounts = FileIO.loadAccountData(System.in); // from some other class in the system
    }

    /** Returns a read-only iterator over the accounts in this collection. */
    public Iterator iterator() {
        return Collections.unmodifiableList(accounts).iterator();
    }
}
```
4. Writing Space
5. JUnit Test Case

Write a JUnit test case class that tests the AccountCollection specified in the previous problem. Your test class should be a valid JUnit test file that has the following properties:

- It should have at least two testing methods that will be run by JUnit.
- It should verify the first precondition from the previous problem; in other words, it should make sure that an IllegalArgumentException is thrown if a null account is added to an AccountCollection.
- It should add at least three accounts to the collection, and then verify that they are present in the collection and are in sorted order (even if the accounts were not added in such an order).
- It should verify the second postcondition from the previous problem; it should make sure that every account's balance is non-negative after calls to applyFees.
- It should verify that the various methods of AccountCollection do not fail when they are called on an empty collection (with no accounts in it).
public class Account {
    public Account(String id, double balance) {
    }
    public void deposit(double amount)
    public double getBalance()
    public String getID()
    public void withdraw(double amount)
}