1. **Software Lifecycle**

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

<table>
<thead>
<tr>
<th>Use case</th>
<th>Record a TV Show</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Level</td>
<td>Summary</td>
</tr>
<tr>
<td>Precondition</td>
<td>NuVo can talk to its program listing database</td>
</tr>
<tr>
<td>Minimal Guarantee</td>
<td>User receives some error message, and failure is logged</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>Show is recorded and put into list</td>
</tr>
</tbody>
</table>

### Main Success Scenario
1. User logs in.
2. User selects show to record.
3. NuVo queues show to record and deducts the time from the account.
4. NuVo records the show and puts it in the list.

### Extensions
1a. Bad user credentials entered.
   1a1. NuVo re-prompts up to 3 times.
2a. User is a child and not permitted to record this show.
   2a1. NuVo shows a failure message and return to the main menu.
2b. User does not have enough minutes in their account to record this show.
   2b1. NuVo shows a message and goes to Purchase Minutes screen.
3a. Queue of shows is full.
   3a1. NuVo bumps lowest show from the queue.
3b. Another show to be recorded already occupies that time slot.
   3b1. NuVo prompts to overwrite or cancel.

## 3. UML Class Diagram

[Diagram of UML Class Diagram showing relationships between GUI, NuVo, ShowList, TV, Genre, Rating, User, Child, PayPerView, Series]
4. Implementation

i:

at (a) add
extends Observable

ii:

at (c) add
if (account == null)
    throw new IllegalArgumentException();

at (b) add
private boolean invariants() {
    if (accounts == null)
        return false;

    for (int i = 1; i < accounts.size(); i++) {
        Account prev = (Account)accounts.get(i-1);
        Account curr = (Account)accounts.get(i);
        if (prev.compareTo(curr) > 0)
            return false;
    }

    return true;
}

at (e), (i) add
assert invariants();

at (c), (f), (i), (j) add
assert accounts != null;

at (e), (h), (i) add
setChanged();
notifyObservers();

at (g) add
assert (account.getBalance() >= 0);
5. JUnit Test Case

public class TestAccountCollection extends TestCase {
    public void testEmpty() {
        AccountCollection c = new AccountCollection();
        try {
            c.add(null);
            fail("null account added");
        } catch (IllegalArgumentException e) {} 

        c.applyFees();
        assertTrue(c.iterator() != null);
        c.load();
    }

    public void testContract() {
        AccountCollection c = new AccountCollection();
        c.add(new Account("b", 2.00));
        c.add(new Account("c", 0.50));
        c.add(new Account("a", 1.00));

        assertTrue(c.isSorted());

        c.applyFees();

        for (Iterator i = c.iterator(); i.hasNext(); ) {
            Account a = (Account)i.next();
            assertTrue(a.getBalance() >= 0);
        }
    }
}