SAMPLE SOLUTION DEMO EXAM

Software Engineering winter term 2011

1. Requirements and Design

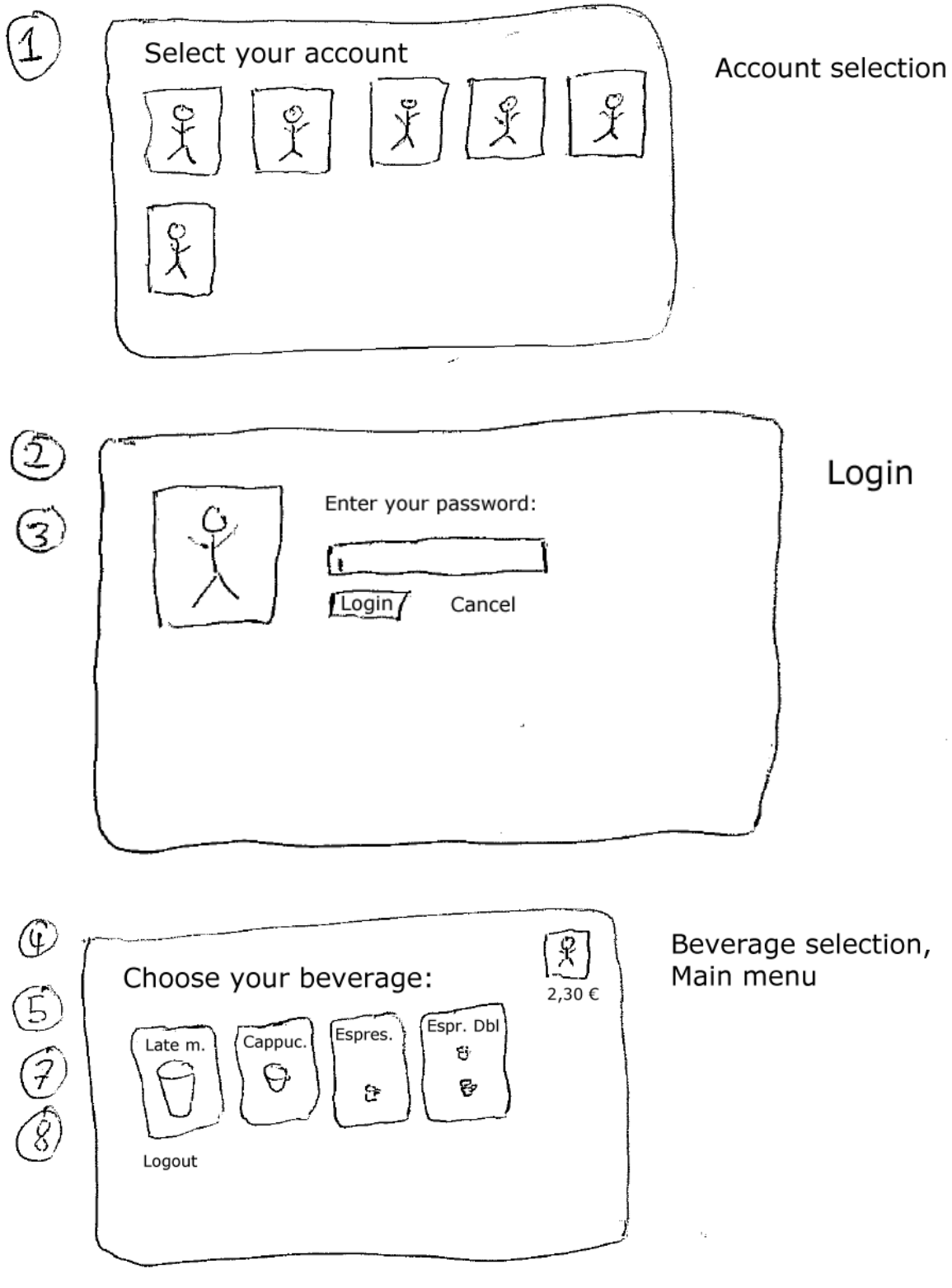
a) Create a use case (in Pressman style) for the above scenario including alternatives and exceptions.

Use Case 1: A student gets a coffee

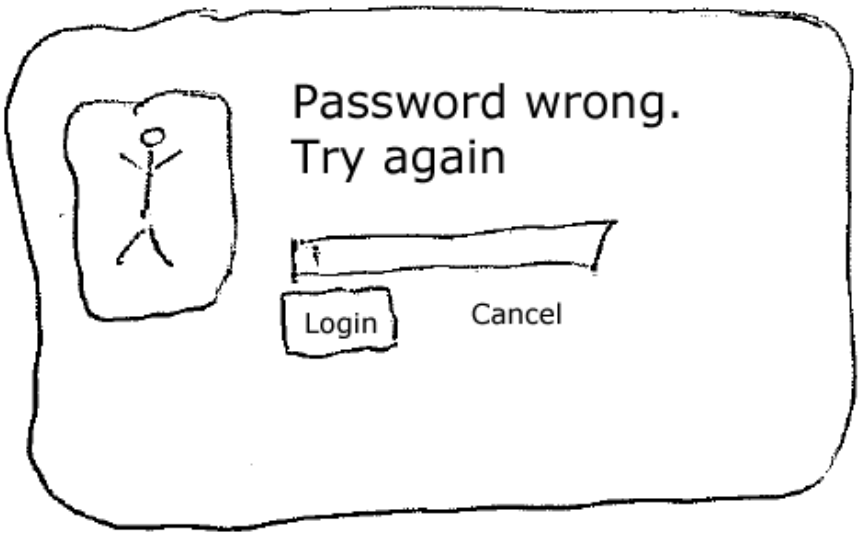
Primary Actor:	Student
Goal in context:	To get a coffee
Preconditions:	1. Student needs to have an account in the system 2. Student needs to have sufficient money on his account 3. Coffee management system is switched on and previous user (if any) has logged out.
Trigger:	Student just had or will have a coffee and wants to pay for it.
Scenario:	1. Student selects his account by clicking on his picture 2. Student enters his password 3. Student clicks on “Login”-Button 4. System shows a list of available beverages to choose from 5. Student chooses the beverage he likes to have by clicking on the respective button 6. Remaining money of student’s account is displayed for 5 seconds 7. System returns to main menu 8. Student clicks on the “Logout”-Button
Exceptions:	1. User enters wrong password: An error message is shown and system returns to step 2. 2. User enters wrong password for fifth time: Error message is shown that the account is blocked and needs to be unblocked by administrator. Contact data to administrator is shown as well. System continues with step 1.
Alternatives:	1. Student doesn’t have enough money for the selected beverage: Message is shown with information that account doesn’t have enough money and on how to increase account money. System continues with step 4.
Priority:	High
Frequency of Use:	Very often - this is the main functionality of the system.

Remark: From the given data it is not defined whether the account has a minimum limit or not. Hence, the use case without precondition 2 and alternative 1 would be valid as well.

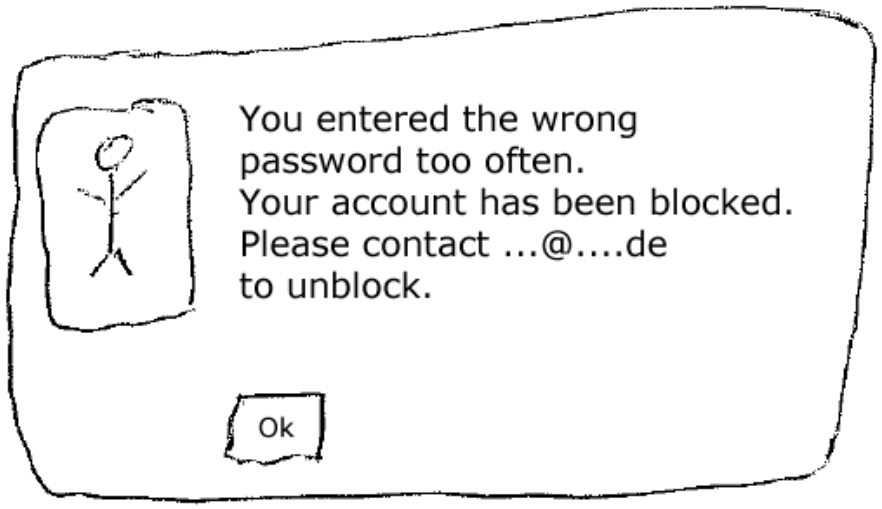
b) Design a user interface for the above use case as a series of screen shot sketches - again, including alternatives and exceptions.



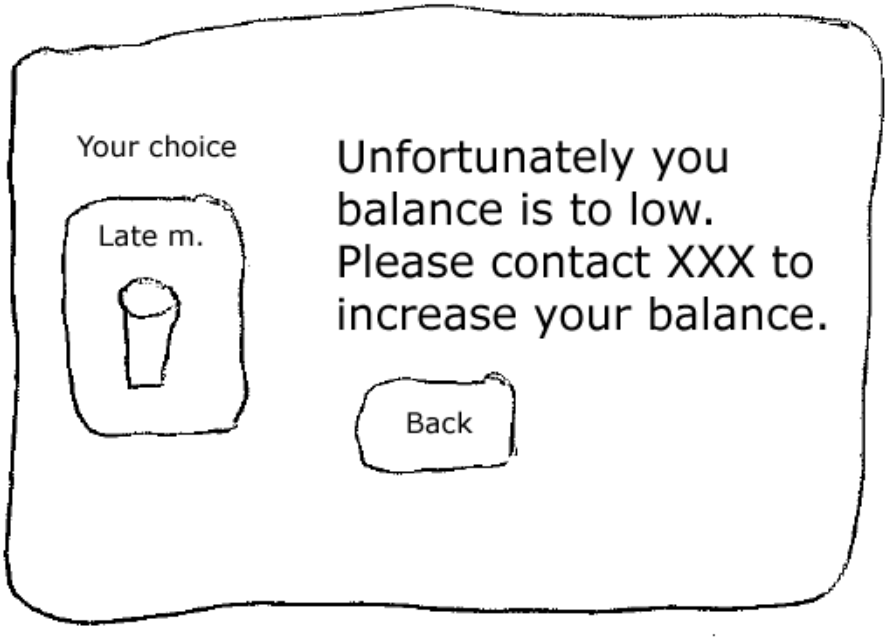
Ex 1


Password wrong.
Try again
Login
Cancel

Ex 2


You entered the wrong
password too often.
Your account has been blocked.
Please contact ...@....de
to unblock.
Ok

AI 1


Your choice
Late m.
Unfortunately you
balance is to low.
Please contact XXX to
increase your balance.
Back

C) Develop a class model for the coffee management system

1. Class-Responsibility-Collaborators

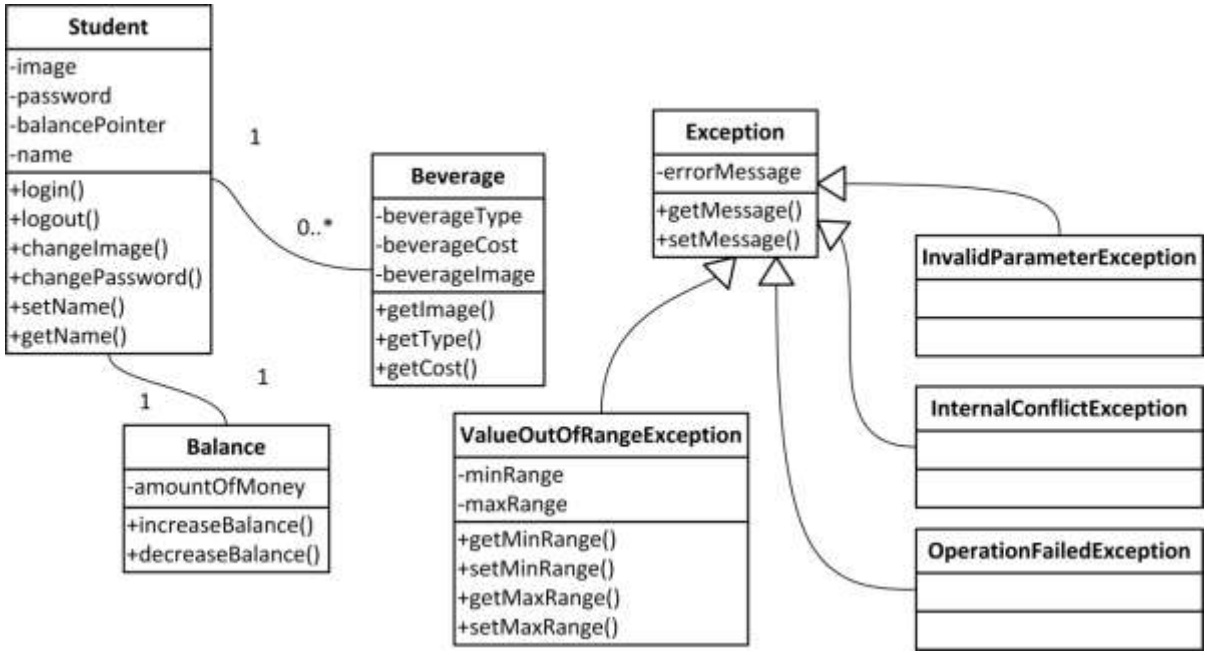
Student	
Responsibilities: - Manage student information (image, password) - Store balance object - Login - Logout	*Collaborators:* - Beverage - Balance - Exception

Beverage	
Responsibilities: - Store information about baverage (price, image)	*Collaborators:* - Student - Exception

Balance	
Responsibilities: - Store amount of available money - Manage decrease and increase of balance	*Collaborators:* - Student - Exception

Exception	
Responsibilities: - Handling of unexpected events	*Collaborators:* - Student - Beverage - Balance

2. UML system model listing



2. Functional Testing

a) Design a set of test cases that achieve *state coverage* with as little effort as possible

To achieve state coverage we need to cover each state out of the set {start, 0, 1, 2} at least once. The only valid solution for this exercise is the following, since all other solutions need more effort:

```
[[<init>, openPort, auth, quit]]
```

The following would be the solution only considering the automata structure:

```
[[<init>, openPort, auth]]          (this is incorrect!)
```

But since it was asked for a sequence of messages and the above is not a complete message, this answer would be wrong.

Hint: The first bracket is starting the sequence of messages – in this case with only one message – the second bracket is indicating the start of the first message.

b) Design a set of test cases that achieve *transition coverage* with as little effort as possible

To achieve transition coverage, we need to use each edge of the graph at least once. The following is a possible solution, covering all edges out of {<init>, openPort, authSend, data, mail, helo, rcpt, authReceive, quit, auth, quit}:

```
[[<init>, openPort, helo, data, mail, quit],
 [<init>, openPort, authSend, rcpt, authReceive, auth, quit]]
```

There are many other solutions, but all of them use the transitions above only in a different order. A valid solution cannot have more transitions.

3. Structural Testing

a) For each of the following coverage criteria, provide an input to *decoded* that achieves 100% coverage:

The given function was exhaustively discussed in the lecture slides "11 – Structural Testing".

1. Statement coverage

To ensure 100% statement coverage, simply each statement of the function needs to be executed at least once. The following input achieves statement coverage:

```
"a+b+%23+%xy"
```

2. Branch coverage

For 100% branch coverage every branch in the control flow graph (CFG) needs to be executed at least once. 100% branch coverage always implies 100% statement coverage (the other way doesn't always hold). For the given program it is not possible to construct an input that has 100% statement coverage but less than 100% branch coverage – therefore we can simply take the input string from above as a solution:

```
"a+b+%23+%xy"
```

What is the difference between statement coverage and branch coverage? Let's consider the following function:

```
int check(bool criterion) {
      if (criterion) {
            std::cout << "criterion is true\n";
      }
      std::cout << "end of method\n";
}
```

For the call

```
check(true);
```

we have 100% statement coverage, since all statements are executed. But still the first decision has only executed to true but not to false – therefore we achieve only 50% branch coverage.

3. Branch and condition coverage

To achieve 100% condition coverage, not only every decision (a Boolean expression composed of conditions and zero or more Boolean operators – normally given as an if-statement) has to be evaluated to both true and false at least once, but also every condition (Boolean expression containing no Boolean operators). Here are some examples to explain the difference between decisions and conditions:

One decision, three conditions:

```
if ((a && b) || c) {...}
```

One decision, one condition:

```
if (3 + 2 == 10 - 5) {...}
```

In the given function, we have only one decision that consists of more than one condition and that is the following:

```
if (digit_high == -1 || digit_low == -1) {
```

Therefore we can simply take the input string we created for 100% branch coverage above and additionally make sure we have 100% condition coverage as well. We get the following input string:

```
"a+b+%23+%ay%xb"
```

4. MC/DC coverage

Achieving 100% MC/DC coverage is a little bit more sophisticated than achieving 100% coverage for the criteria above.

So let us first make clear what 100% MC/DC coverage means. For *condition coverage*, each condition of a decision has to be at least once true and once false. This can lead to two necessary runs for a decision with many conditions therefore missing many possible combinations. On the other hand, trying all possible combinations (also called Multiple condition coverage) leads to a combinatorial explosion of 2^n runs for n conditions. For 6 conditions, we would already need 64 runs. So either we lose some tests which might be important or we'll have an exploding runtime.

MC/DC coverage tries to solve this dilemma by simply executing the *important combinations* to avoid exponential blowup. Now which combinations are important? Simple: The ones in which conditions actually influence the result of the decision.

This leads us to the following: For 100% MC/DC coverage

- Each **decision** has taken the value true and false at least once (branch testing)
- Each **condition** has taken the value true and false at least once (condition testing)
- Each condition in the decision **independently affects** the decision's outcome

Now let's go back to the given function. We have only one decision with more than one condition, so let's investigate that one:

```
if (digit_high == -1 || digit_low == -1) {...}
```

We'll first create a truth table for the conditions of the decision:

```
A: digit_high == -1
B: digit_low  == -1
```

A	B	Result
True	True	True
True	False	True
False	True	True
False	False	False

Now we extend the table such that it indicates which test cases can be used to show the independence of each condition:

#	A	B	Result	A	B
1	True	True	True		
2	True	False	True	4	
3	False	True	True		4
4	False	False	False	2	3

To show the independence of A we need test case 2 and 4.
To show the independence of B we need test case 3 and 4.
The resulting minimal set of test cases therefore is: {2, 3, 4}.

Thus we need to ensure the following:

```
digit_high == -1,   digit_low != -1
digit_high != -1,   digit_low == -1
digit_high != -1,   digit_low != -1
```

This can be achieved using the following input:

```
"a+b+%23+%ay%xb%ab"
```

Hint: If you feel you need a more detailed explanation on MC/DC, I'd like to recommend the talk "MC/DC in a nutshell" by Christopher Ackermann which can be found here: http://www.cs.umd.edu/~atif/Teaching/Fall2006/StudentSlides/MCDC_Coverage_02.ppt

5. Loop boundary coverage

For achieving 100% loop boundary coverage, we need to execute each loop

- Exactly zero times
- Exactly one time
- More than once

The following three test cases ensure 100% loop boundary coverage:

{"", "a", "abc"}

4. Mixed Bag

1. Functional testing tests against the specification
True

2. Structural testing attempts to find missing functionality
Wrong

Structural tests are based on the code structure and cover the implemented behavior. Missing functionality is not implemented and will therefore not be tested by structural testing.

3. The OO hierarchy principle defines team structure
Wrong

The OO hierarchy principle introduces inheritance which is the ability for a *class* to extend or override functionality of another *class*. It has nothing to do with team structure.

4. Weyuker's hypothesis states that coverage criteria are only intuitively defined.
True

5. The V model is an important design pattern.
Wrong

The V model is a systems development model. It tries to simplify the understanding of the development of complex systems.