

6.170: Quiz 1 Solutions

- Don't open this booklet until you are directed to do so.
- Write your name on the front page.
- Circle your TA's name and your section number.
- This quiz is 50 minutes long. It contains **7** questions and has **12** pages. Please check your copy to make sure it is complete before you start.
- Good luck!

Problem	Grade	Points
1		6
2		10
3		15
4		15
5		20
6		4
7		30
Total		100

Circle your TA and Section #:

Section 1: Matt Section 2: Roshan
Section 3: Kalpak Section 4: Brandy
Section 5: Allen Section 6: Todd
Section 7: Kurt Section 8: Godfrey
Section 9: Jon

Name: _____

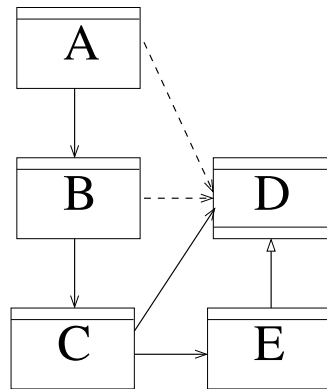
1 True/False (6pt)

Read each statement carefully. Decide if the statement is true or false and then circle your answer.

- 1.1 True / False The “depends” relation is transitive.
- 1.2 True / False Preconditions should always be avoided because they are a burden to the clients using the code.
- 1.3 True / False Operational specifications are preferred over declarative specifications because the client needs to know the specific steps involved in the implementation of a method.
- 1.4 True / False An observer method of a type will never modify the representation of the type.
- 1.5 True / False Representation exposure is desirable in the design of an abstract data type, so the client can use methods of the representation not exposed through the ADT.
- 1.6 True / False The representation invariant should hold at all times during the execution of a method.

2 Module Dependency Diagrams (10pt)

Examine the MDD and answer the following questions.



Which parts need to be examined (answer with A, B, C, D, and/or E)...

2.1 If only the specification of D.method1() is changed? **C, E**

2.2 If only the implementation of E.method1() is changed? **none**

2.3 If D is replaced with a new part F? **A, B, C, E**

3 Specifications (15pt)

3.1 Write a complete declarative specification of the method `removeDuplicates`, whose function is to remove duplicates from List `l`. Do not assume default values for *requires*, *modifies* or *effects*. Feel free to write your specification in plain English.

```
public static void removeDuplicates(List l) {
    if ((l == null) || (l.size() == 0))
        return;

    List copy = new ArrayList(l);
    Iterator elements = copy.iterator();
    Object a = elements.next();

    while (elements.hasNext()) {
        Object b = elements.next();
        if (a.equals(b))
            l.remove(b);
        else
            a = b;
    }
}
```

requires: `l` is sorted (or `l` is arranged so that cards with equal values are adjacent)
modifies: `l`
effects: If the list passed in is null or has no elements, then it is unchanged.
Otherwise, the list has all duplicates removed.

4 Revealing Subdomains (15pt)

Examine the following snippet of buggy code from the `TwoPair` class for the method `evaluateHand`, then answer the question on the next page.

```
1  class TwoPair extends PokerRanking { // ...
2      /**
3       * Analyzes the specified poker hand to determine whether it can
4       * be ranked as a TwoPair. If the specified hand is invalid or
5       * cannot be associated with this ranking, the method returns
6       * null.
7       *
8       * ***IMPORTANT NOTE***
9       * A valid TwoPair ranking is defined as any hand which has
10      * AT LEAST two pairs of cards. Two cards constitute a pair if
11      * their values (CardValue) are equal.
12      *
13      *
14      * @param h the poker hand to be evaluated
15      * @return a TwoPair poker ranking if it can be associated with
16      *         the specified hand; null otherwise
17      */
18     public static PokerRanking evaluateHand(Hand h) {
19
20         if(!isValidHand(h))
21             return null;
22
23         List handCopy = new Vector();
24
25         // lists the cards in this hand, ordered from lowest card to highest
26         // (Aces are treated as greater than Kings)
27         Iterator acesHighIterator = h.listCardsAcesHigh();
28
29         // make a copy of all cards in the hand
30         while(acesHighIterator.hasNext())
31             handCopy.add(acesHighIterator.next());
32
33         Card c1 = (Card)handCopy.get(0);
34         Card c2 = (Card)handCopy.get(1);
35         Card c3 = (Card)handCopy.get(2);
36         Card c4 = (Card)handCopy.get(3);
37         Card c5 = (Card)handCopy.get(4);
38
39         // check for two pairs
40         if(c1.equals(c2) && c3.equals(c4) && !(c4.equals(c5))) {
41             SortedSet remainingCards = new TreeSet();
42             remainingCards.add(c5);
43             return new TwoPair(c1, c3, remainingCards);
44         }
45         else {
46             // did not find two pairs
47             return null;
48         }
49     }
50     // ... ...
51 }
```

4.1 To develop a test suite for this method, you must find the revealing subdomains for the input space (the set of all Hands). Identify **THREE** possible revealing subdomains and give one sample input Hand from each such subdomain. You may assume all Hands are valid (i.e., consist of 5 cards) and have at least two pairs.

Hint: Your three subdomains and their corresponding sample inputs must be distinct after line 27.

There were two possible answers to this question based on what assumption you made about the `Card.equals` method.

- Assuming `Card.equals` tested equality of only values of Cards and ignored Suits:

Sample inputs from revealing subdomains:

Suit doesn't matter. We only indicate the value of each card in the Hand. Note the trick that the hand is ordered (since we call `h.listCardsAcesHigh()`, all inputs are ordered before we read the "if" conditions.)

1. 3 3 5 9 9 - All valid TwoPairs in which the extra card is third.
2. 3 5 5 9 9 - All valid TwoPairs in which the extra card is first.
3. 3 3 5 5 5 OR 3 3 3 5 5 - A Full House (still a valid TwoPair based on our definition)

Invalid answers:

1. 3 3 5 5 9 - All valid TwoPairs in which the extra card is last and not equal to the first. Invalid answer because result is correct for such inputs.
2. 2 3 4 5 6 OR 2 3 3 4 5 - These are obviously not a TwoPair and hence not a revealing subdomain since the method will return the right answer (null).

- Assuming `Card.equals` tested true equality of both Values and Suits:

Full credit was given to those who stated this assumption and gave three valid and distinct two-pair examples. Full credit was also given to those who only provided one example but **EXPLAINED** their reasoning for providing just one example (essentially any valid TwoPair hand).

Sample inputs from revealing subdomains:

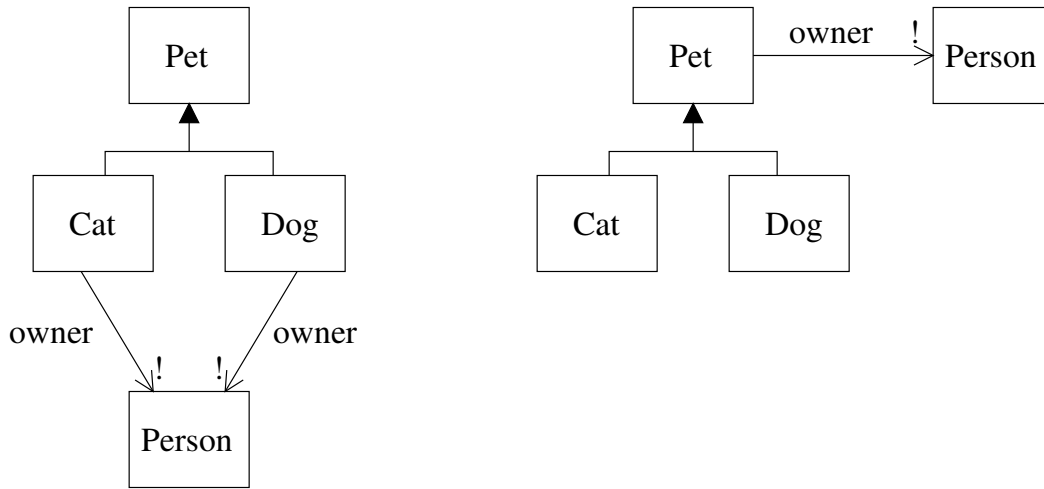
1. 3H 3S 5H 9S 9H - All valid TwoPairs in which the extra card is third.
2. 3H 5S 5H 9S 9H - All valid TwoPairs in which the extra card is first.
3. 3H 3S 5H 5S 5C - A Full House

Invalid answers:

1. 2H 2H 2H 2H 2H - Any hand in which you explicitly used duplicate cards (same value and same suit). A game of Poker is only played with one deck, hence you cannot have identical cards, so these cannot be part of your input space.
2. 2H 3H 4H 5H 6H - These are obviously not a TwoPair and hence not a revealing subdomain since the method will return the right answer (null).

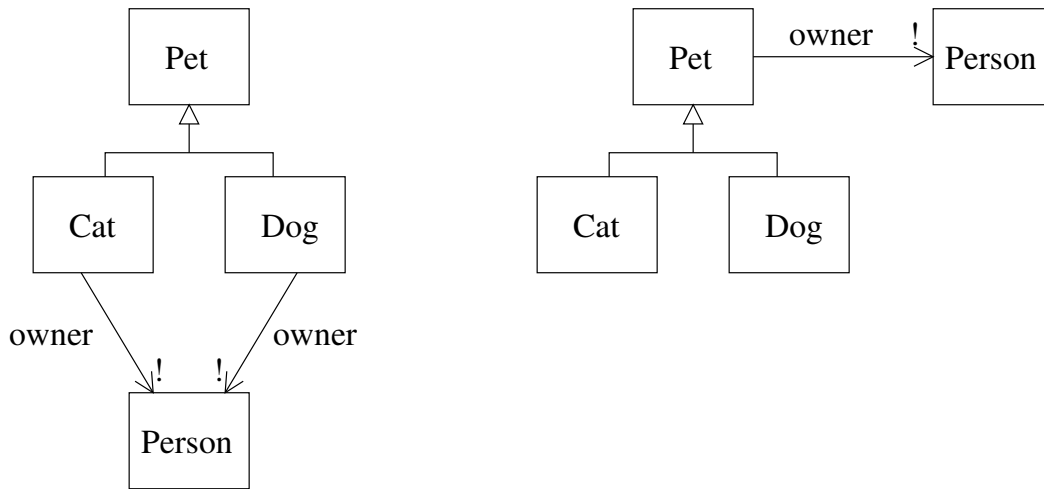
5 Object Models (20pt)

5.1 Are these two OMs equivalent? Explain your answer.



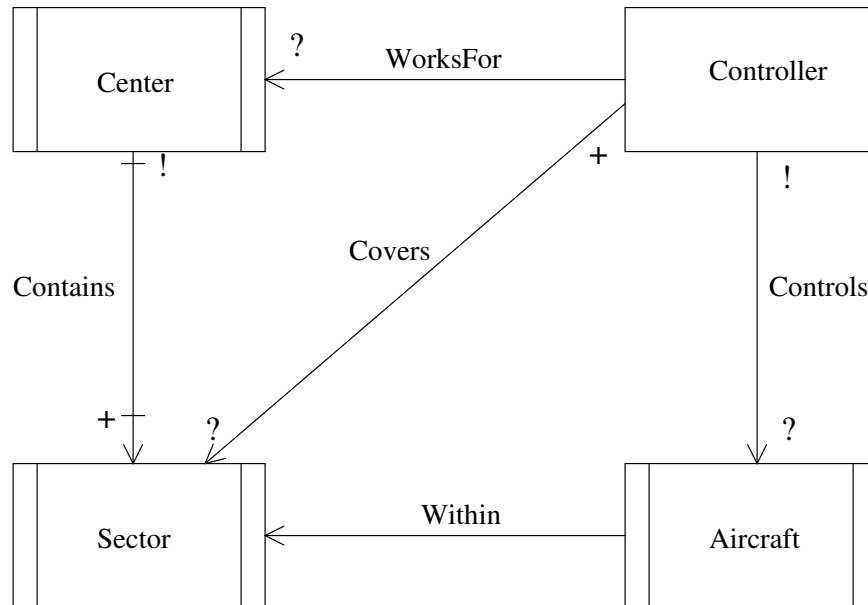
Yes. Cat and Dog together make an exhaustive subset of Pet, and they each have owners.

5.2 Are these two OMs equivalent? Explain your answer.



No. There may be another Pet, Snake, which has no owner in the left diagram, but it must have an owner in the right diagram.

5.3 In the object model below, we have the sets *Controller*, *Aircraft*, *Sector* and *Center*. Fill in the multiplicity and mutability markings according to Rules 1-6.



1. Controllers work for at most one Center.
2. An Aircraft is controlled by exactly one Controller.
3. A Controller may cover a Sector, and a Sector is covered by one or more Controllers.
4. Centers contain one or more Sectors and each Sector is in exactly one Center.
5. The Centers and Sectors are fixed.
6. The Sector to Center mapping cannot change: one Sector cannot be moved to another Center and a Center cannot contain new Sectors.

6 Exceptions (4pt)

For each snippet of code, indicate whether the exception being thrown should be checked or unchecked and explain why.

6.1

```
if (!checkRepInv())  
    throw new AException();
```

Unchecked. Not preserving the rep invariant is a failure of the program.

6.2

```
//requires: x >= 0 public void insert(int x) { if (x < 0) throw new  
CException(); }
```

Unchecked. Not meeting the precondition is a general failure of the program.

7 Abstraction Functions and Rep Invariants (30pt)

Examine the following ADT for a stack, then write its representation invariant, abstraction function, and sketch an inductive proof for the representation invariant.

```
// A NonEmptyStack is a sequence of Integers with one or more elements.
class NonEmptyStack {
    Vector elements;
    int stackLength;

    // modifies: this
    // effects: Creates a new NonEmptyStack with
    // firstElement as the first stack element
    public NonEmptyStack(Integer firstElement) {
        elements.add(firstElement);
        stackLength = 1;
    }

    // modifies: this
    // effects: push newElement into stack
    public void push(Integer newElement) {
        if(elements.size() > stackLength) {
            elements.set(stackLength, newElement); // set elements[stackLength] = newElement
        } else {
            elements.add(newElement); // append newElement to elements
        }
        stackLength++;
    }

    // modifies: this
    // effects: If the stack has more than one element,
    // remove the top element. Otherwise, do nothing
    public void pop() {
        if(stackLength != 1)
            stackLength--;
    }

    // effects: Return the top element of the stack.
    public Integer top() {
        return (Integer) elements.get(stackLength-1);
    }
}
```

7.1 Abstraction Function:

A sequence of Integers from `elements[0]` ... `elements[stackLength-1]` where `elements[stackLength-1]` is the top element of the stack.

7.2 Representation Invariant (complete the following clause):

All elements are Integers && ...
`elements.size() ≥ stackLength` &&
`stackLength > 0`.

7.3 Give an inductive proof that the Representation Invariant holds through the process of stack creation and any sequence of calls to push, pop and top: Base case: From constructor, `stackLength = 1`, `elements.size() == 1`, all elements in elements are Integers

Hypothesis:

1. `stackLength > 0`
2. all elements in elements are Integers
3. `elements.size() ≥ stackLength`

push:

Claim: Invariant 2 holds:

Two cases:

1. `(elements.size() > stackLength) -->`
`stackLength ≤ elements.size()`
2. `(elements.size() ≤ stackLength) &&`
`(elements.size() ≥ stackLength) // from hypothesis 2`
`--> elements.size() == stackLength`

Then, because `elements.add()` increments vector size and we execute `length++`, we get `elements.size() == stackLength` right before we exit `push()`.

Hence, Invariant 2 holds.

Finally,

1. we only increment `stackLength` so Invariant 3 holds
2. we only add Integers so Invariant 1 holds.

pop:

Claim: Invariant 2 and 3 holds:

Two cases:

1. `(stackLength != 1) &&`
`(stackLength > 0) // from hypothesis 1`
`--> stackLength ≥ 2 --> stackLength ≥ 1 after pop() exits.`
2. `(stackLength == 1) &&`
`--> stackLength == 1 after pop() exits`

Also, elements do not change so Invariant 1 holds.

top:

Just an observer method without benevolent side effect.

The end!