

Massachusetts Institute of Technology  
6.170 Laboratory in Software Engineering

Spring 2004

Quiz 2

97

Monday, April 5, 2004

Name: DAN PORTS

Athena User Name: drkp

Section (check one):

- |  |   |   |
|--|---|---|
| <input type="checkbox"/> 1. Matt Notowidigdo         | <input type="checkbox"/> 4. Brian Dunagan | <input type="checkbox"/> 7. Gregory Marton    |
| <input type="checkbox"/> 2. Magdalena Balazinska     | <input type="checkbox"/> 5. Cliff Frey    | <input type="checkbox"/> 8. Paul Pham         |
| <input checked="" type="checkbox"/> 3. Michael Bolin | <input type="checkbox"/> 6. Ryan Jazayeri | <input type="checkbox"/> 9. Asfandyar Qureshi |

**Instructions**

Do not open this booklet until instructed to do so.

This quiz is closed book, closed notes. You have **50 minutes** to complete it. It contains 14 questions in 11 pages (including this page), totaling 100 points. Before you start, please check your copy to make sure that it is complete. Turn in all pages, together, when you finish.

Write your recitation section number and your initials at the top of each page.

Please write neatly; no credit will be given if we cannot read what you write.

Good luck!

1. True/false questions. Circle T if the statement is true, F if not.  
[2 points each, 14 points total]

- T  F The best person to perform black-box testing on a module is the original programmer, since she understands the code's potential weaknesses.
- T  F The abstraction function for a class depends on its representation.
- T  F A class designed to be extended by others may need to include details of its implementation in its specification.
- T  F One way to improve the usability of a user interface is by removing features.
- T  F The observer design pattern is used to separate the methods of a class into observers and mutators.
- T  F If we declare that class A extends B, Java will check that class A meets the specification for class B.
- T  F During user testing, subjects should be given specific objectives to achieve, but not told how to achieve them.

2. [4 points] Constructors in Java have some limitations that can be overcome with design patterns.

a. Constructors in Java can never return an instance of a subtype of the class to which they belong. Name a design pattern that avoids this limitation.

FACTORY ✓

b. Constructors in Java can never return a pre-existing object. Name a design pattern that permits such reuse. Your answer should be different to part (a).

SINGLETON ✓

18/18

Your Section #: 3

Your Initials: DRK i

Please read the specifications below for an interface called Door and a class called LockableDoor. The questions on the next 3 pages refer to these specifications.

*overview: Door represents a door in a building. A Door is mutable.*

```
public interface Door {  
  
    public void open() throws CannotOpenException;  
        modifies: this  
        effects: the door is open, or CannotOpenException is thrown if door can't open  
  
    public void close();  
        modifies: this  
        effects: the door is closed, or does nothing if door can't close  
  
    public Room lookInside() throws DoorClosedException;  
        effects: returns room on other side of door, or throws DoorClosedException if  
        door is closed  
  
}
```

*overview: LockableDoor represents a Door with a lock.*

```
public class LockableDoor implements Door {  
  
    public LockableDoor(Room room, Key key);  
        effects: makes a LockableDoor for room, initially locked with the given key  
  
    public void unlock(Key key) throws WrongKeyException;  
        modifies: this  
        effects: unlocks door if the door was locked with the given key, or throws  
        WrongKeyException if the door was locked with a different key  
  
    public void open() throws CannotOpenException;  
        modifies: this  
        effects: the door is open, or CannotOpenException is thrown if door is locked  
  
    public void close();  
        modifies: this  
        effects: the door is closed - or does nothing if door can't close  
  
    public Room lookInside() throws DoorClosedException;  
        effects: returns room on other side of door, or throws DoorClosed Exception if  
        door is closed.  
  
    public boolean equals(Object obj);  
        effects: returns true if other object is behaviorally equivalent to this one  
  
}
```

3. [9 points] LockableDoor is a true subtype of Door. For each of the following changes (considered separately), indicate by ticking the appropriate box whether LockableDoor is still a true subtype of Door after the change.

a. Change Door to a class; LockableDoor now extends Door.

- still a subtype  
 no longer a subtype

b. Change Door's open() method so that it no longer throws the exception CannotOpenException. Add a precondition to LockableDoor's open() method requiring that the door be unlocked.

- still a subtype  
 no longer a subtype

c. Change LockableDoor's open() method so that it takes a Key as a parameter.

- still a subtype  
 no longer a subtype

9/9

4. [8 points] The specification for lookInside() throws an exception if the door is closed. Give two alternative specifications for Door's lookInside() that deal with this case differently, without throwing exceptions.

(a) @ returns room on other side of door, or null if door is closed

8/8

(b) public static Room DOOR\_CLOSED = ...  
@ returns room on other side of door, or DOOR\_CLOSED if door is closed.

5. [5 points] Should WrongKeyException be a checked exception or an unchecked exception? Why?

Checked. Unlocking a door with the wrong key is an exceptional condition that should be anticipated and handled by the client code. + no way to check if key is ok without calling unlock()

4/5

Here is another class implementing Door. Unlike the descriptions of Door and LockableDoor, this is not just a specification; it includes method bodies as well.

*overview: UnlockedDoor represents an unlocked Door.*

```
public class UnlockedDoor implements Door {  
  
    private LockableDoor door;  
  
    public UnlockedDoor(LockableDoor door, Key key)  
        throws WrongKeyException {  
        effects: makes an UnlockedDoor by unlocking the door with the key,  
            or else throws a WrongKeyException if it's the wrong key  
        door.unlock(key);  
        this.door = door;  
    }  
  
    public void open() throws CannotOpenException {  
        modifies: this  
        effects: opens door, or throws CannotOpenException if door can't open  
        door.open();  
    }  
  
    public void close() {  
        modifies: this  
        effects: the door is closed - or does nothing if door can't close  
        door.close();  
    }  
  
    public Room lookInside() throws DoorClosedException {  
        effects: returns room on other side of door, or throws DoorClosedException if  
            door is closed.  
        return door.lookInside();  
    }  
}
```

6. [4 points] What design pattern is used for the relationship between UnlockedDoor and LockableDoor? (Several answers are possible here; giving one is enough.)

adapter

4

7. [6 points] Louis Reasoner wants us to change the implementation of UnlockedDoor to make unlocking a lazy operation, so that if the door is never opened, we never have to unlock it. What is wrong with this proposal, if the specification of UnlockedDoor has to stay the same?

If the door is not unlocked until we open it, we do not know whether the key is correct until we try to open it. Thus open() would need to throw a WrongKeyException, which the spec doesn't allow. 6

8. [6 points] Complete UnlockedDoor's equals method below so that it implements behavioral equivalence:

```
public boolean equals(Object o) {
    effects: returns true if and only if this and o are behaviorally equivalent
```

```
    if (!(o instanceof UnlockedDoor)) return false;
    UnlockedDoor d = (UnlockedDoor) o;
```

```
    // add your code here
```

```
    return (door.equals(d.door));
```

6

9. [9 points] Louis Reasoner makes the following false statements. Explain why they're false.

a. "Black-box tests are sufficient for unit testing a module."

3 There may be hidden revealing subdomains that result from implementation details and cannot be deduced from the spec.

E.g. a method that looks up small results in a table, and computes larger ones,

b. "As long as you build a system from the bottom up, you'll never need to write a testing stub."

1 Writing a testing stub can make it easier to write unit tests for top-level components, since the values returned from the testing stubs can be directly specified. It also makes it possible to unit test top-level classes without depending on the correctness of bottom-level classes.

c. "Assertions are too expensive to keep in released code, so only use them when you're testing and debugging."

3 Some assertions do not noticeably increase the code's runtime. These should be left in the released code, so that internal inconsistencies can still be caught before they lead to unexpected behavior.

10. [7 points] Read the code below. For each of the assertions, mark whether the assertion is:

- **False** (for at least one execution that is valid according to the spec)
- **Useless** (the assertion should be true, but it is not worth checking)
- **Good** (the assertion should be true, and is worth checking)

```
private static boolean isSorted(List list);
requires: all elements in list are mutually comparable by the Comparable interface
returns: true if and only if list is in sorted order, so that list[0] ≤ list[1] ≤ ... ≤ list[n-1]

private static void split(List list, List a, List b);
modifies: a, b
effects: splits list into two halves and puts first half into a and second half into b;
thus list = a' + b', and the lengths of a' and b' differ by at most 1 element

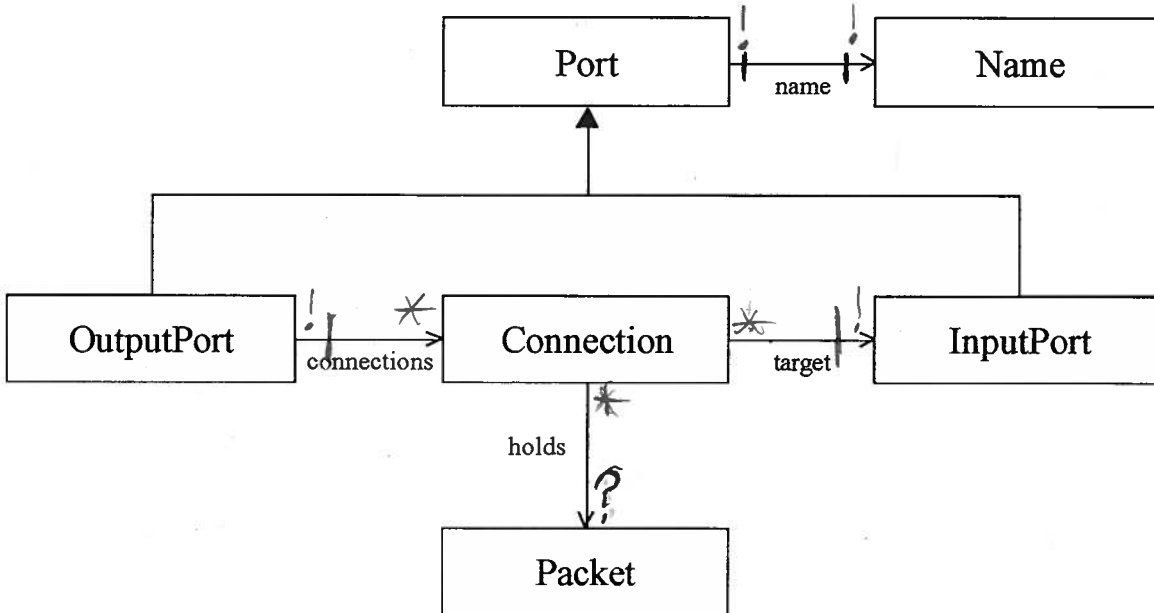
private static void merge(List list, List a, List b);
modifies: list
effects: concatenates a and b into list, so that list' = a + b
```

```
public static void mergeSort (LinkedList list) {
requires: all elements in list are mutually comparable by the Comparable interface
modifies: list
effects: list is put in sorted order, so that list'[0] ≤ list'[1] ≤ ... ≤ list'[n-1]
state whether each assertion is "False (F)", "Useless (U)", or "Good (G)" on left
```

|                 |  |
|-----------------|--|
| ↓<br>✓ <u>F</u> | <code>assert(!isSorted(list));</code>  |
| ✓ <u>U</u>      | <code>if (list.size() &gt;= 2) {</code><br><code>  assert(!list.isEmpty());</code>   |
| ✓ <u>U</u>      | <code>  List a = new LinkedList();</code><br><code>  List b = new LinkedList();</code><br><code>  assert(a != null &amp;&amp; b != null);</code> |
| ✓ <u>G</u>      | <code>  split(list, a, b);</code><br><code>  assert(list.size() == a.size() + b.size());</code>  |
| ✓ <u>U</u>      | <code>  mergeSort(a);</code><br><code>  mergeSort(b);</code><br><code>  assert(isSorted(a) &amp;&amp; isSorted(b));</code>                       |
| ✓ <u>G</u>      | <code>  merge(list, a, b);</code><br><code>  assert(list.size() == a.size() + b.size());</code>  |
| ✓ <u>G</u>      | <code>  }</code><br><code>  assert(isSorted(list));</code><br><code>}</code>   |

*. redundant  
if list  
assertion is  
in place*

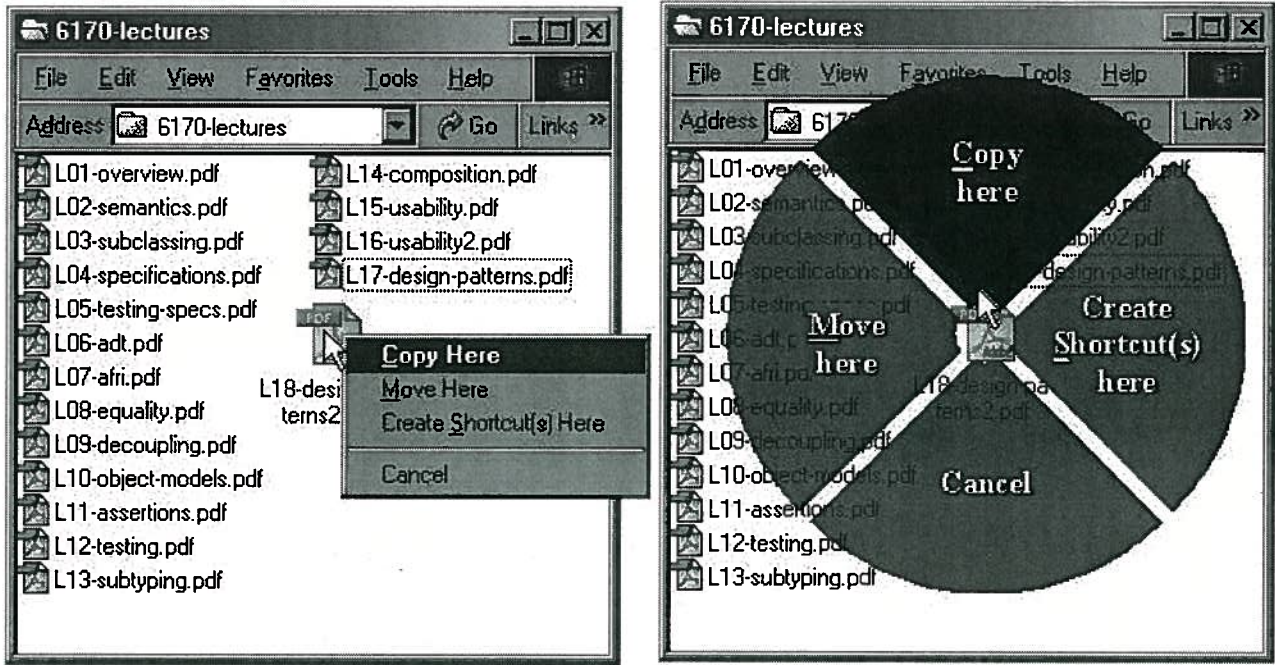
11. [9 points] The object model below is a representation of a message-passing system. Add **multiplicity** and **mutability** markings to the arrows of the diagram to capture the constraints listed below. You do not need to show which markings correspond to which constraints.



Constraints:

- ✓ ▪ Every Port has a unique Name that never changes.
- ✓ ▪ A Connection is created from one OutputPort to one InputPort, and it continues to connect those Ports until it is destroyed. Connections can be created or destroyed at any time.
- ∩ ▪ OutputPorts and InputPorts can have any number of Connections.
- ∩ ▪ A Connection holds at most one Packet at a time.
- ✓ ▪ When an OutputPort needs to send a message, it checks which of its Connections are free (holding no Packets), and creates a single Packet that is placed on all the free Connections. No Packet is created if no free Connections are found.

12. [8 points] Ben Bitdiddle is designing a menu that pops up when the user presses the right mouse button. He's considering two alternatives: the conventional rectangular menu shown on the left, or the circular *pie menu* shown on the right.



By appealing to Fitts's Law ( $T = a + b \log(D/S + 1)$ ), give two reasons why the pie menu would generally be faster to use than the rectangular menu.

- (a) The pie slices are larger than the rectangular menu items. Therefore, for the pie, the size  $S$  is larger and the time  $T$  to acquire the correct position is smaller.
- (b)

All pie slices are equally close to the cursor's initial position, while the bottom menu items of the rectangular menu are further away. So the distance  $D$  is larger for the rectangular menu, and  $T$  is higher.



13. [6 points] Louis Reasoner is extending his Traffick user interface. To guarantee that the user always clicks on streets, he writes some code like this:

```
assert (onStreet (mousePosition));
```

Give two reasons why this assertion is a bad idea, in terms of the user's experience.

- (a) If the assertion fails, the program will terminate. This causes a simple user error to lead to an unnecessarily bad result. It would probably be reasonable to just ignore the mouse click.
- (b) The only error message produced by this code will be something like "assertion failed", which is entirely unhelpful to the user. It does not indicate what happened or how to avoid or correct the error.

14. [5 points] Louis Reasoner tries to convince Ben Bitdiddle that the only methodical way to develop software is in alphabetical order. He proposes to write code, specifications, and tests in the order shown on the left below. But Ben prefers test-driven development. Please list the same tasks in the order that Ben performs them (you may write just the first letter of each task).

**Task-list for Louis**

- (B) Black-box tests
- (C) Code
- (G) Glass-box tests
- (R) Regression tests
- (S) Specifications

**Task-list for Ben**

- \_\_\_\_\_ S
- \_\_\_\_\_ B
- \_\_\_\_\_ C
- \_\_\_\_\_ G
- \_\_\_\_\_ R

END OF QUIZ

11/11