iMassachusetts Institute of Technology
6.170 Laboratory in Software Engineering

Spring 2004

Quiz 2

Monday, April 5, 2004

Name: ____

Athena User Name: _________________

Section (check one):

___ 1. Matt Notowidigdo	___ 4. Brian Dunagan	___ 7. Gregory Marton
___ 2. Magdalena Balazinska	___ 5. Cliff Frey	___ 8. Paul Pham
___ 3. Michael Bolin	___ 6. Ryan Jazayeri	___ 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.

False. The code may make unwarranted assumptions, and if the coder writes the tests those assumptions may remain untested.

T F The abstraction function for a class depends on its representation.

True. If the representation of a class changes, so must its abstraction function, though not the space of abstract values into which it projects.

T F A class designed to be extended by others may need to include details of its implementation in its specification.

True. For example, skeletal implementations may need to include self-call information for a client to predict the effects of overriding a method.

T F One way to improve the usability of a user interface is by removing features.

True. Removing features reduces the complexity of a UI.

T **F** The observer design pattern is used to separate the methods of a class into observers and mutators.

False. Pattern is concerned with notifying dependents when an object changes.

T **F** If we declare that class A extends B, Java will check that class A meets the specification for class B.

False. Java knows nothing about preconditions, postconditions, etc.

T F During user testing, subjects should be given specific objectives to achieve, but not told how to achieve them.

True. You want to see how they try to achieve the goal, and shouldn't prompt 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 (or singleton)`________________

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).

`_interning (or factory, singleton,…)_`

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.

- [X] still a subtype
- [ ] no longer a subtype

LockableDoor still meets specification for Door

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
- [X] no longer a subtype

LD.open can no longer be used everywhere that D.open could be

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

- [ ] still a subtype
- [X] no longer a subtype

LD now does not have an implementation of the method D.open()

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)

add a requires clause -
requires: door is open, effects: returns room

(b)

use a special return value
e.g. *effects: returns room on other side of door, or null if door is closed*

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

It should be a checked exception. With the current interface, it is hard for a client to be sure the key is correct before calling unlock. If she could, an unchecked exception would be reasonable, since if would be annoying to force her to try/catch even if she is sure there will be no exception. As she can't avoid the possibility of this exception, it is good to remind her to catch this outcome, which should not be catastrophic.

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.)

composite, any wrapper

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?

The specification for UnlockedDoor constructor promises to throw an exception if the key is the wrong key for the door. The only way to find out if the key is right is by calling unlock(), so we can't delay doing that.

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);


}
```

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."

Black-box tests may not exercise all sections of code. We need clear-box tests to do that.

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

We need stubs to simulate external entities the system interacts with, e.g. network.

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

Expense is context-dependent. An O(1) assertion in an O(n) method probably isn't expensive. It is useful to leave such assertions in since we want to detect bugs at the earliest point possible even in released code.

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]

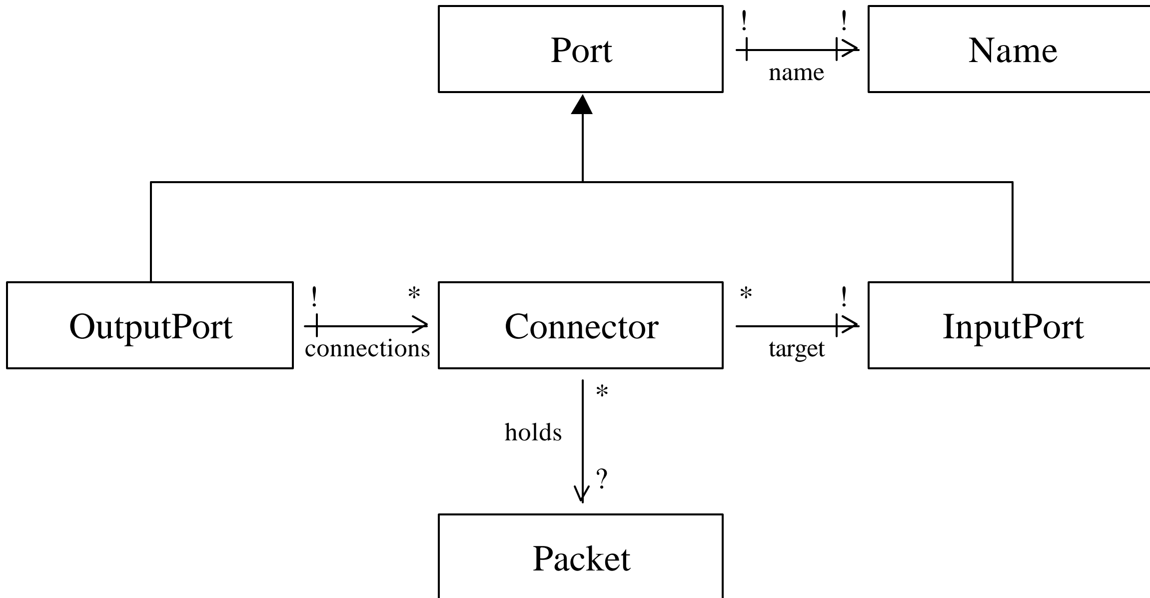
```
private static void split(List list, List a, List b);
```
modifies: a, b
effects: splits list into two halves and puts first half into a and second half into b; thus list = a' + b', and the lengths of a' and b' differ by at most 1 element

```
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

```
  __F__      assert(!isSorted(list));

             if (list.size() >= 2) {
  __U__          assert(!list.isEmpty());

                 List a = new LinkedList();
                 List b = new LinkedList();
  __U__          assert(a != null && b != null);

                 split(list, a, b);

  __G__          assert(list.size() == a.size() + b.size());

                 mergeSort(a);
                 mergeSort(b);

  _G/U_          assert(isSorted(a) && isSorted(b));

                 merge(list, a, b);

  __G__          assert(list.size() == a.size() + b.size());
             }

  __G__      assert(isSorted(list));
         }
```

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.

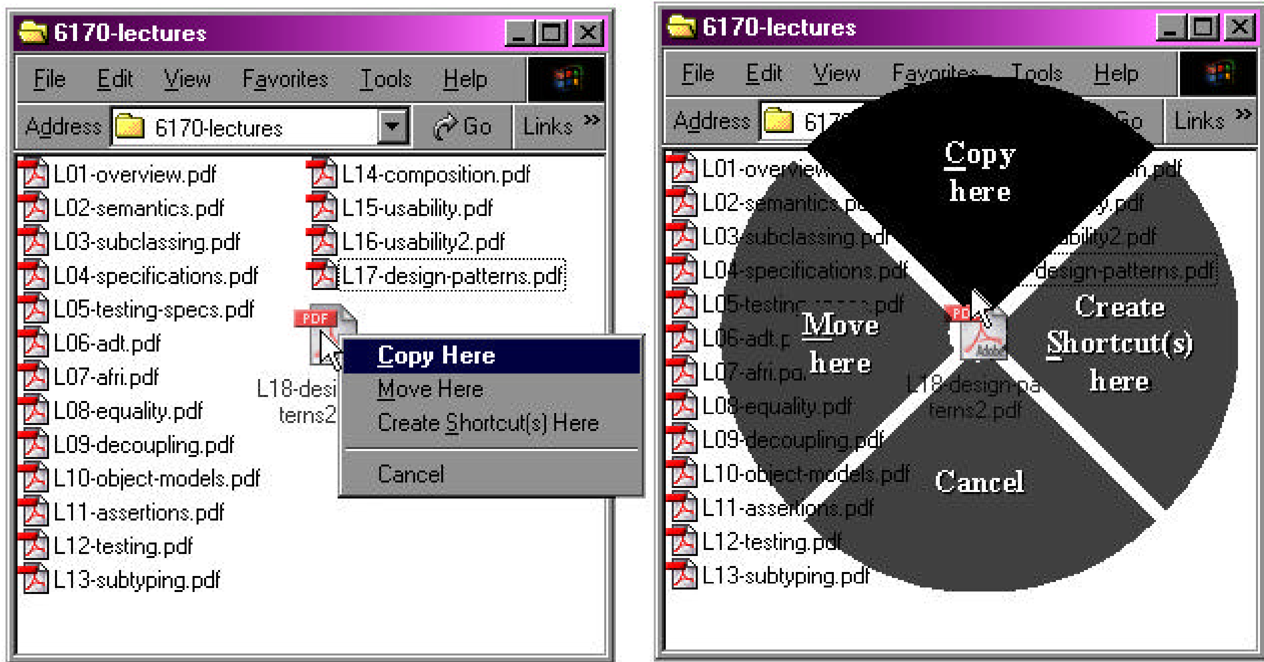


Constraints:

- 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.

One valid solution is given in the figure. Full marks were given for any solution consistent with the constraints.

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)

All options on pie menu are equidistant and close to mouse, so $D_{pie} < D_{rect}$.

(b)

The size of the target depends on the radius of the pie, which can be very large (maybe extending out to edge of screen) so $S_{pie} > S_{rect}$.
So we expect that $T_{pie} < T_{rect}$.

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)

It should be okay for user to click in wrong place without crashes.

(b)

The assertion will give an error message that is not helpful to user.

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		**Task-list for Ben**
(B) Black-box tests	*first task*	_Specifications________________
(C) Code	*second task*	_Black-box tests______________
(G) Glass-box tests	…	_Code____________________________
(R) Regression tests		_Glass-box tests______________
(S) Specifications		_Regression tests_____________

END OF QUIZ