Massachusetts Institute of Technology
6.170 Laboratory in Software Engineering

Fall 2001 QUIZ: Wednesday, October 31, 2001

Name: *Solutions*

Athena username: ______________________

Section (circle one):

Sect 1, Nii Dodoo	Sect 2, Brendan Kao	Sect 3, Brandy Leung	Sect 4, Michal Mirvis
Sect 5, Kurt Steinkraus	Sect 6, Jonathan Whitney	Sect 7, Robert Lee	

This quiz is 50 minutes long. It contains 73 questions and 11 pages (excluding this one). Please check your copy to make sure it is complete before you start. You may separate this sheet from the rest of the test, and just turn in the first sheet. Mark **T** or **F** in each box. Blank boxes will not receive any credit. Incorrect answers will not be penalized.

True/False - Part One

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

True/False - Part Deux

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33

Casting, Specifications

34	35	36	37	38

Object Models

39	40	41	42

Design Patterns

43	44	45

Substitution Principle

46	47	48	49

Template Methods

50	51	52	53

Skeletal Impl.

54	55	56	57

Optional Methods

58	59	60	61

Command Pattern

62	63	64	65

Java Interfaces

66	67	68	69

Rep Exposure

70	71	72	73

True/False - Part One [1 point each]

1. Industrial software averages 0.1 bugs per thousand lines of code.
   False. Industrial software averages about 10 bugs/kloc.

2. Top-down design is an effective way of achieving decoupling of modules.
   False. Top-down design involves splitting a module into subparts, developing them, and then combining them together.

3. If there is no code reuse, then the uses graph is a tree.
   True. Reuse would cause a part to have multiple users.

4. The uses relationship (as described in lecture) is transitive.
   True. If A ***is affected by*** B ***and*** B ***is affected by*** C***, then*** A ***is affected by*** C***.***

5. A weak dependence of A on B means that A depends only on the name of B.
   True. A ***weakly depends on*** B ***if*** A ***refers to*** B ***by name but doesn't make use of any service it provides.***

6. Every Object has a `hashCode` method.
   True. Every class extends `Object` ***and therefore inherits all of its methods (including*** `hashCode`***).***

7. In the Java API, for mutable objects, the `equals` method and == are equivalent.
   False. The `equals` ***method tests for observational equivalence while == tests for behavioral equivalence.***

8. Observer methods may mutate the representation.
   True. An observer may have benevolent side effects. Observers may mutate the rep, so long as the abstract value is preserved.

9. It is good practice to maximize inherited code since that maximizes code reuse.
   False. Inheritance adds extra dependences that complicate the MDD, the design, and the implementation, making it harder to code, understand, and modify.

10. A problem object model can only represent a finite set of configurations.
    False. A problem object model can represent an infinite number of legal configurations.

11. Given a code object model, one can deduce all dependences.
    False. There are some aspects that cannot be expressed graphically.

12. A precondition can simplify the implementation of a method.
    True. The most common use of preconditions is to demand a property because it would be hard or expensive to check.

13. Methods of immutable types always have empty modifies clauses in their specifications.
    True. If a type is immutable no method can modify it.

14. Java's access control and strong typing provide representation independence.
    True. Java's access control mechanisms allow you to control dependences. Controlling access to the fields of a class helps give representation independence. Strong typing ensures that an access that is declared to be to a value of type t ***in the program text will always be an access to a value of type*** t ***at runtime.***

15. The abstraction function is a many-to-one mapping that maps every concrete value to an abstract value.
    False. Concrete values that are not allowed by the representation invariant will not have a mapping to an abstract value.

16. Abstraction functions depend on the representation of a type, not just the specification.
    True. Different representations have different abstraction functions.

True/False - Part Deux [1 point each]

17. Operational specifications are preferable because they do not expose implementation details.
False. Operational specifications describes how something is done – not the same as pre / post conditions.

18. It is good practice to always check the precondition and throw an appropriate exception if the precondition has been violated.
False. Precondition can be much too expensive to check each time method is called.

19. Using immutable types is not efficient because they cannot be shared.
False. Immutables can not be altered hence there is no danger in sharing them.

20. The strongest possible representation invariant may not include all properties of the object model.
True. For example, in the exercise to create a Graph, it is possible that the OM stated that a Node is part of one and only one Graph. Yet, this is not something that either the rep. invariant of Graph or Node can check.

21. Asserting postconditions tends to catch bugs earlier than asserting representation invariants.
False. Figure presented in Lecture.

22. When iterating over a `Collection` of objects using an `Iterator`, one should not modify the `Collection`.
NOT GRADED. Question meant to ask if the modification to the `Collection` ***directly rather than through the*** `Iterator`***. In this case, the answer is true, since modifying the*** `Collection` ***will invalidate the*** `Iterator`***.***

23. In an object model, a ! at the source end of an arrow from A to B means that an A object points to exactly one B object.
False. It means that B belongs to exactly one A for the relation represented by the arrow.

24. It is easy to ensure that the equals method defines a reflexive relation, but harder to ensure that its symmetric or transitive.
True. An implementation of equals that is reflexive can simply be a comparison using the java "==".

25. The representation of an ADT is always exposed if one of its methods returns a mutable object.
False. Returning a mutable object in the rep does expose the rep. See page 111 of the book.

26. The representation invariant must hold at the start of every method of a class.
    False. Not true for private methods.

27. A mutable ADT may have an immutable rep.
    False. If it has immutable rep, the ADT can't be mutable.

28. An immutable ADT may have a mutable rep.
    True. Changing the internal representation does not mean a client can detect anything different through observors. For example, an ADT can restructure its internal data upon each observer access in order to optimize for lookups. Yet, there is no visible difference to the client.

29. The purpose of testing is to prove program correctness.
    False. Testing can reveal the presence of errors but never their absence.

30. Method/Procedure-level specifications are only meaningful in the context of an Object Oriented programming language.
    False. Even non OO programming languages can have procedures – and these procedures need specifications about their behavior.

31. It is possible to expose mutable member fields of an object without risking rep exposure.
    True. It is only risking rep exposure when the mutable member field can be modified by a client so as to break the rep invariant.

32. Java supports multiple inheritance of specifications only.
    True. Java does not support multiple inheritance of classes (inheritance of code), but it does support multiple "implementations" of interfaces (inheritance of specification).

33. The implementation of every class should be annotated with a representation invariant.
    False. A class can have no private fields.

Casting, Specifications [2 points each]

```
Object a = map.get(x) ;
Vector v = (Vector) a ;
```

34. The cast may change the static (declared) type of the object returned by the `get` method.
False. Casting does not change the type (static or dynamic) of an object. It is merely an instruction to the type checker that the programmer knows that a certain variable has a more specific type than is statically verifiable.

35. The cast may change the dynamic (run-time) type of the object returned by the `get` method.
False. Casting does not change the type (static or dynamic) of an object. It is merely an instruction to the type checker that the programmer knows that a certain variable has a more specific type than is statically verifiable.

Consider the code:

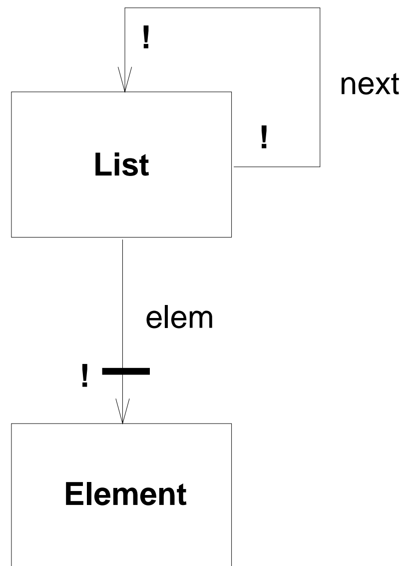
```
class Logger {

   // requires: msg!= null
   // effects: prints msg to this logger's output stream
   public boolean log(String msg) { ... }

}
```

36. A better design would eliminate the precondition in the specification of `log` .
True. If the `log` ***method handles the fast null case check by not outputting anything, then this relieves some of the burden on client code to do null-checking.***

37. The postcondition is underspecified.
True. The effects: clause should specify the meaning of the return value of the `log` ***method.***

38. A better design would make the `log` method static.
False. The `Logger` ***class maintains some state on the output stream and so it cannot be made a static method.***

Object Models [2 points each]



This object model describes a representation of a linked list in whch the List objects form the nodes of the list and the Element objects are the elements contained. A list implementation developed according to this object model:

39. can represent an empty list.
False. The **!** ***marking on the*** **elem** ***relation mandates that every element of List is related to exactly one element of Element.***

40. cannot be implemented without mutation.
True. Because of the self-edge in List, this is a circular singly linked-list. Therefore, when you add a new node, it is necessary to mutate existing nodes to preserve the circluar nature.

41. allows elements of the list to be replaced without changing the linking between nodes.
False. The mutability marking on the **elem** ***relation does not allow replacing of list elements.***

42. allows duplicate elements.
True. The elem relation relates each List to exactly one Element, and each Element to zero or more List's.

Design Patterns [2 points each]

Consider the following pieces of code that embody design patterns.

```
Example 1:

    Factory f = new BikeFactory();
    f.makeBike();

Example 2:

    class Gym {
        private static Gym theGym;
        private Gym() { }
        public static Gym getGym() {
           if (theGym == null) theGym = new Gym();
           return theGym;
        }
    }
```

43. Example 1 uses the factory method design pattern.
    False. It corresponds to the factory object design pattern.

44. Example 2 is an example of a flyweight design pattern.
    False. The flyweight design pattern is a generalization of interning used for mostly immutable objects.

45. Example 2 is an example of a singleton design pattern.
    True. The singleton design pattern guarantees that only one object of a particular class exists.

Substitution Principle [2 points each]

46. If class `A` is a subtype of class `B` (according to the subsitution principle), then class `A` is a legal Java subclass of `B`.
    False. The subsitutition principle allows the arguments of a method of `A` ***to be supertypes of the arguments of the corresponding method of*** `B`***, but Java does not allow this if*** `A` ***is to be a subclass of*** `B`***.***

47. If class `A` is a legal Java subclass of class `B`, then class `A` is a subtype of `B` (according to the substitution principle).
    False. The substitution principle requires that behavioral properties of the specification of `B` ***be satisfied by*** `A`***, which is not required in Java subclassing.***

48. The substitution principle says that `A` is a subtype of `B` if the specification of `A` is stronger than the specification of `B`.
True. The above defines the substitution principle.

49. The substitution principle says that `A` can be a subtype of `B` if `A.foo()` throws more exceptions than `B.foo()`.
False. `A.foo()` ***has to throw the same exceptions as*** `B.foo()`.

Template Methods [2 points each]

50. Template methods are a key part of framework implementations in object-oriented languages.

    True. We discussed the 'Hollywood Principle' in lecture: don't call us we'll call you. The template method is the basis for this style, with calls to hook methods that get overridden.

51. Template methods may make calls to hook methods that contain no code.
    True. Without extensions, the framework doesn't need to do anything.

52. Template methods are always overridden in subclasses.
    False. It's the hooks that are usually overridden, not the templates.

53. Template methods are used in JUnit to implement common stages of a test.
    True. The template method calls hook methods setUp, runTest, tearDown.

Skeletal Implementations [2 points each]

The skeletal implementations of the Java Collections API

54. make it easy to implement basic versions of sets, lists and maps.
    True. That's what they're for, in addition to factoring out shared code in the Java API.

55. have no data representations of their own, but rely on subclasses to provide them.
    True. `AbstractList` ***assumes a method to get the i-th element, for example, then builds an iterator, equality method, etc, on top of this.***

56. are implemented with template methods.
    True. The template methods are the substance of the code; they work by calling hook methods defined in a subclass.

57. are abstract classes that should be used by clients in preference to concrete classes when declaring collection variables.
    False. That's what the interfaces are for. See discussion in notes on preferring interfaces to abstract classes. This is perhaps the key aspect of the Java API design, and one well worth emulating.

Optional Methods [2 points each]

58. Optional methods are a special feature of Java that allows a class implementing an interface to declare only some of the methods.
False. They're a stylistic feature of the API, and nothing to do with the language. An implementing class must always declare the methods of an interface.

59. Optional methods should throw the UnsupportedOperationException if the method is not actually implemented.
True. Otherwise clients of the class may fail in mysterious ways.

60. Optional methods help control the proliferation of types.
True. This was the motivation for optional methods that was explained in lecture.

61. Optional methods are sometimes used in the Java Collections API to restrict the functionality of views.
True. A more subtle point. The `keySet` ***view of a map, for example, does not implement the*** `add` ***method.***

Command Pattern [2 points each]

62. The command pattern is central to the organization of Tagger.
True. Actions are commands. The use of the command pattern is so central and extensive it defines the entire style of the program, providing much of its flexibility and elegance, but also causing some obscurity and complexity of hidden interaction.

63. The command pattern can often be conveniently implemented using anonymous inner classes.

    True. See the code of Tagger. Discussed briefly in lecture.

64. The command pattern is used for test cases in JUnit.
True. This was our first encounter with the command pattern; in addition to the lecture notes, there's an explanation in the Cook's Tour article.

65. The command pattern is a pattern that underlies listeners in graphical user interfaces.
False. That's Observer, not Command.

Java Interfaces [2 points each]

66. A Java interface is usually preferable to an abstract class because concrete classes can be more easily retrofitted to it.
True. Discussed in lecture on Java Collections API.

67. A Java interface cannot declare static methods.
True. One of the key limitatons of interfaces.

68. A Java interface helps decoupling by eliminating dependences of client code on concrete classes.
True. Topic of second lecture on decoupling.

69. A Java interface cannot be used in a typecast, since every object's type is a concrete class.
False. Typecasts are just runtime type checks, so it makes no difference whether the type corresponds to a concrete class or not.

Rep Exposure [2 points each]

70. Mutable objects may cause rep exposure if used as keys in a Java Hashmap.
True. A defect of the Java approach to equality. Discussed in lecture on equality, copying and views.

71. A program with only immutable types cannot suffer from rep exposure.
True. Rep exposure means breaking the rep invariant by a side effect, and if there aren't any side effects, the problem goes away.

72. Java's conventions for equality make rep exposure more likely.
True. See lecture on equality, copying and views. This is one respect in which the Liskov approach is preferable.

73. A type with a rep invariant of true cannot suffer from rep exposure.
True. There's no rep invariant to worry about.