University of Washington
CSE 331 Software Design & Implementation
Spring 2012

Final exam

Monday, June 4, 2012

Name: *Solutions*

CSE Net ID (username):

UW Net ID (username):

This exam is closed book, closed notes. You have **110 minutes** to complete it. It contains 27 questions and 16 pages (including this one), totaling 220 points. Before you start, please check your copy to make sure it is complete. Turn in all pages, together, when you are finished. **Write your initials on the top of *ALL* pages** (in case a page gets separated during test-taking or grading).

Please write neatly; we cannot give credit for what we cannot read.

Good luck!

Page	Max	Score
2	8	
3	12	
4	20	
5	12	
6	22	
7	26	
8	20	
9	16	
10	12	
11	24	
12	10	
13	10	
14	8	
15	5	
16	15	
Total	220	

1 True/False

(2 points each) Circle the correct answer. T is true, F is false.

1. $\boxed{\textbf{T}}$ **/ F** Top-down testing typically requires the tester to build method stubs.
2. **T /** $\boxed{\textbf{F}}$ Top-down testing typically requires the tester to build test drivers.
3. **T /** $\boxed{\textbf{F}}$ For a class that represents an ADT (which excludes some GUI classes, for example), the Javadoc should always include an Abstraction Function.

   The Javadoc is public documentation, which should not reveal implementation details.
4. $\boxed{\textbf{T}}$ **/ F** A user interface that looks aesthetically beautiful may actually be bad in terms of usability.

2 Multiple choice

(3 points each) Mark the single best choice, by circling the appropriate letter.

5. A design pattern used to enhance the functionality of an object is
   (a) Adapter
   [✓ (b)] Decorator
   (c) Delegation
   (d) Proxy

6. A design pattern often used to restrict access to an object is
   (a) Adapter
   (b) Decorator
   (c) Delegation
   [✓ (d)] Proxy

7. You have a class that accepts and returns values in British Imperial units (feet, miles, etc.), but you need to use metric units. The design pattern that would best solve your problem is
   [✓ (a)] Adapter
   (b) Decorator
   (c) Delegation
   (d) Proxy

8. Which of the following class relationships best fits the composite pattern?
   (a) A `Zoo` contains a `Set<Exhibit>`, an `Exhibit` contains a `Set<Animal>`, and an `Animal` contains a number of properties about that individual animal. To get information about a particular `Animal`, a client would write something such as:
   `Zoo.getExhibit("Penguins").getPenguin("Tux").getAge();`
   (b) `Dalmatian` is a subclass of `Dog`, which is a subclass of `Mammal`, which is a subclass of `Animal`. Each subclass overrides some methods while using the inherited version of others, for some shared behavior and some distinct behavior.
   (c) `GeometricShape` is an interface implemented by `Square`, `Circle`, `Sphere`, and `Dodecahedron`. Though they have the same public interface and can all be used anywhere a GeometricShape is required, they otherwise have no relationship and do not depend on each other.
   [✓ (d)] The class `Food` is implemented by `PeanutButterAndJellySandwich`, which contains objects of type `Bread`, `PeanutButter`, and `Jelly`. `Bread` contains `Flour` and `Salt`, and `Jelly` contains `Fruit` and `Sugar`. All of these objects are `Food` objects themselves.

(5 points each) Mark all of the following that can be true, by circling the appropriate letters.

9. Suppose that you change a specification by removing a precondition and adding/modifying/removing some other clause, such as a throws clause. The new specification might be:

   [✓ (a)] stronger

   [✓ (b)] weaker

   [✓ (c)] incomparable

   [✓ (d)] same strength (i.e., equivalent)

   Full credit if at least A and C; B and D are ignored.

10. Which of the following is a use case supported by standard version control systems?

    [✓ (a)] Managing several versions or releases of a software program

    (b) File bug reports and track their progress

    [✓ (c)] Allowing team members to work in parallel

    [✓ (d)] Identifying when and where a regression occurred

11. Which of the following are facts about a top-down implementation approach?

    (a) A top-down process is more time consuming because of the unit tests.

    [✓ (b)] Top-down lets you present a demo of the project to the management faster than using a bottom-up process.

    (c) In a top-down design, if an error is detected it's always because a lower-level module is not meeting its specifications (because the higher-level ones are already been tested).

    (d) A top-down process makes it possible to detect performance problems faster

    [✓ (e)] A top-down process makes it easier to fix a global conceptual problem

12. Which of the following are appropriate uses of assert statements? Assume that all of this is application code — it is not a fragment of a unit test.

    (a)
    ```
    int oldSize = myList.size();
    myList.add(element);
    assert myList.size() == oldSize+1;
    ```

    (b)
    ```
    assert myList.add(element);
    ```

    [✓ (c)]
    ```
    /** @requires element != null */
    public void add(E element) {
      assert element != null;
    ```

    [✓ (d)]
    ```
    public void add(E element) {

      ...
      assert repOK(); // like checkRep(), but returns a boolean
      return true;
    ```

    A is false because it clutters the code. B is false because it has side effects.

3 Fill in the table

13. (12 points) Suppose you have a program P. Consider the following statements about a given test suite. Write ⇒ in the following table to indicate which statements imply which other ones? For example, if statement **b** implies statement **c**, you would write ⇒ in the **b row** and the **c column**.

    (a) The test suite was created using the revealing-subdomain method, and the partitions were chosen perfectly with respect to P.

    (b) The test suite has 100% statement coverage for P.

    (c) The test suite has 100% path coverage for P.

    (d) The test suite detects all errors in P.

	a	b	c	d
a	⇒			⇒
b		⇒		
c		⇒	⇒	
d				⇒

Partitions can depend on the specific problem. It is not necessary for a test suite to cover both statements in

```
if (p) {
  x = x + x;
} else {
  x = x * 2;
}
```

14. (22 points) Consider the following code.

```
class A {
  void m₁(A x) { System.out.println("AA"); }
  void m₂(B x) { System.out.println("AB"); }
  void m₃(C x) { System.out.println("AC"); }
}
class B extends A {
  void m₁(A x) { System.out.println("BA"); }
  void m₂(B x) { System.out.println("BB"); }
  void m₃(C x) { System.out.println("BC"); }
}
class C extends B {
  void m₁(A x) { System.out.println("CA"); }
  void m₂(B x) { System.out.println("CB"); }
  void m₃(C x) { System.out.println("CC"); }
}

A a1 = new A();
A a2 = new B();
A a3 = new C();
B b1 = new B();
B b2 = new C();
C c1 = new C();
```

Fill in each box with the output of the corresponding method invocation. For example, fill in the 4th row and 2nd column with the output of `b1.m(a2)`.

(Hint: this problem goes pretty fast once you see the pattern.)

	a1	a2	a3	b1	b2	c1
a1	AA	AA	AA	AB	AB	AC
a2	BA	BA	BA	BB	BB	BC
a3	CA	CA	CA	CB	CB	CC
b1	BA	BA	BA	BB	BB	BC
b2	CA	CA	CA	CB	CB	CC
c1	CA	CA	CA	CB	CB	CC

This problem involves both overloading and overriding.

First, notice that there are three distinct method families, which we have labeled $\mathtt{m}_1$, $\mathtt{m}_2$, ***and*** $\mathtt{m}_3$ ***above. These are totally distinct — they have nothing to do one another, except that they happen to have the same name `m`. At compile time, the compiler determines which method family each call in the source code corresponds to. It uses the method name and the signature (the formal parameter types) to do so. For example, consider the call `a2.m(a3)`: because the declared (compile-time) type of `a3` is `A`, this is a call to the method family*** $\mathtt{m}_1$.

Each method family has three overriding implementations. At run time, the JVM uses dynamic dispatch to choose one of the implementations. It uses the class (i.e., the run-time type) of the receiver to do so. For the call `a2.m(a3)`, the receiver has run-time type `B`, so the implementation $\mathtt{B.m}_1$ ***is invoked, and it prints "`BA`".***

Zero credit if at least half of the answers are wrong, since that indicates you do not understand the concepts.

15. Consider the following code. Circle “OK” or “error” to indicate which assignments are type-correct and which are compile-time errors.

    (Hint: when type-checking wildcards, the Java type-checker does not take account of information such as that `Object` has no supertypes and `Double` has no subtypes.)

```
List<Object> lo;
List<? extends Object> leo;
List<? super Object> lso;

List<Number> ln;
List<? extends Number> len;
List<? super Number> lsn;

List<Double> ld;
List<? extends Double> led;
List<? super Double> lsd;

ln = lo;          //  OK / [✓ error]
ln = leo;         //  OK / [✓ error]
ln = lso;         //  OK / [✓ error]
ln = ln;          //  [✓ OK] / error
ln = len;         //  OK / [✓ error]
ln = lsn;         //  OK / [✓ error]
ln = ld;          //  OK / [✓ error]
ln = led;         //  OK / [✓ error]
ln = lsd;         //  OK / [✓ error]

len = lo;         //  OK / [✓ error]
len = leo;        //  OK / [✓ error]
len = lso;        //  OK / [✓ error]
len = ln;         //  [✓ OK] / error
len = len;        //  [✓ OK] / error
len = lsn;        //  OK / [✓ error]
len = ld;         //  [✓ OK] / error
len = led;        //  [✓ OK] / error
len = lsd;        //  OK / [✓ error]
```

16. You find 4 versions of a function that copies the first n elements from List src to List dest.

    ```
    void partialcopy(List<Integer> dest, List<Integer> src, int n)
    ```

    Fortunately, all the implementations have specifications written in CSE 331 style.

    Specification A
    @requires: $n > 0$
    @modifies: dest
    @throws: ArrayOutOfBoundsException if src.size() $< n$
    @effects: for i=1..n, $\text{dest[i]}_{post} = \text{src[i]}_{pre}$

    Specification B
    @requires: $n > 0$
    @modifies: src, dest
    @throws: ArrayOutOfBoundsException if src.size() $< n$
    @effects: for i=1..n, $\text{dest[i]}_{post} = \text{src[i]}_{pre}$

    Specification C
    @requires: $n > 0$ and src.size() $>= n$
    @modifies: dest
    @throws: nothing
    @effects: for i=1..n, $\text{dest[i]}_{post} = \text{src[i]}_{pre}$

    Specification D
    @requires: $n > 0$
    @modifies: dest
    @throws: nothing
    @effects:
    for i=1..min(n, src.size()), $\text{dest[i]}_{post} = \text{src[i]}_{pre}$
    and for i=src.size()+1..n, $\text{dest[i]}_{post} = 0$

    In the following diagram, draw an arrow from X to Y if and only if X is stronger than (implies) Y.

$$\begin{tikzcd}
A \arrow[r] \arrow[d] & B \\
C & D \arrow[l]
\end{tikzcd}$$

    In the interest of reducing code size, which versions of the method can you discard (and replace any uses of them by uses of remaining, non-discarded versions)? Circle all the redundant versions that you can **discard**.

    (a) A
    [✓ (b)] B
    [✓ (c)] C
    (d) D

    Answer: B and C because in both cases there is a stronger version of the method that can be substituted. We graded relative to your answer from the previous part: we looked at whether your answer was consistent with the arrows you drew on the diagram, regardless of whether the arrows were actually correct.

4 Short answer

17. Suppose that your program contains modules A and B, and information flows from A to B. Recall that it is possible for the program dependencies (as expressed in a Module Dependency Diagram, for example) to be either A→B or B→A.

    Write one word each to distinguish the design expressed in these MDDs: (2 points each)

    A→B ***push***

    B→A ***pull***

    Suppose that you are trying to decide which of the two designs to implement. Give two criteria that would guide your decision, and how. Give two criteria that are as different as possible from one another. (1–2 sentences each.) (6 points each)

    (a) **Relative** ***frequency of updates: actual vs. needed. Use pull if the producer produces updates faster than the consumer needs it. Use push if the consumer needs to be updated as soon as any information is available from the producer.***

    Simply "frequency of updates" is not correct.

    (b) ***Future changes. Determine which module is more likely to be replaced by a different/changed implementation (assume it satisfies the same specification), and make that one depend on the other module. Or, determine which modules you would like to reuse.***

    (c) Reduce coupling elsewhere in the MDD by making the choice of direction for this arrow.

    A weak answer is: Amount of information. You need to use **both** ***push (to notify) and pull (to send information) if the producer provides much more information than the consumer needs.***

18. (4 points) State the most important similarity between an interface and an abstract class. (1 sentence)

    Neither one can be instantiated.

19. (4 points) State a circumstance in which you would prefer an interface over an abstract class. (1 sentence)

    When specifying one of multiple supertype behaviors, since a class can implement many interfaces.

20. (4 points) State a circumstance in which you would prefer an abstract class over an interface. (1 sentence)

    When giving default behavior for a set of related data abstractions, since an interface cannot provide a method implementation.

21. (6 points) State a disadvantage of the enumeration design pattern, as it is built into the Java language (the `enum` keyword). (Hint: consider the advantages of the alternatives presented in section.) (1 sentence)

    It uses more space than the alternative of using an `int`.

    It does not permit subtyping, which would be desirable if, for example, one enum's values are a subset of another enum's values. But subtyping can be viewed as a negative, because then a client doesn't necessarily know how many elements a given enumeration has.

    Incorrect answers are that an enum cannot store data, or have methods, or have a natural ordering (it's available via `compareTo`). Another incorrect answer is that the names might be poorly-chosen (that's true for any variable)

    The advantages of the enumeration design pattern almost always outweigh this disadvantage. For example, using `int` constants is much more error-prone.

22. (12 points) Why are paper-and-pencil sketches of architectures, module dependency diagrams, and APIs preferred over software prototypes in the early stages of design? Give at least three reasons. Give reasons that are as different from one another as possible. (Hint: analogize to user interface design.) (1 sentence each.)

    (a) ***It provides less detail, permitting the software engineer to focus on the most important decisions first. A related benefit is that it better visualizes the design than textual code does.***
    (b) ***Cost: The drawings are cheap to create and cheap to change. Thus, designs and re-designs are fast and affordable. (Many students listed this one point as two or even three of their answers.)***
    (c) ***Attachment to code: It's psychologically easy to throw out a paper mock-up. Once code is written, it's much harder to convince oneself to throw it out, even if it is flawed.***
    (d) Your manager, or a reviewer, will view the design as malleable and subject to criticism; source code may give the impression of finality.
    (e) The design can be evaluated without being bound to implementation choices.

    Incorrect answers included anything about UI, layout, and visuals. Also, decomposition and parallel work, and simulation, are as easy or easier in code.

23. (6 points) In 1–2 sentences, explain the circumstances under which a specification should refer to a field that is defined by the implementation. For brevity, give your answer for *either* a class specification or a method specification, but not both. Indicate which one your answer is about.

    Circle one: **class / method** specification. ***Either.***
    Explanation: ***Never. A specification should refer only to specification fields.***

    It's possible that some specification fields are in 1-to-1 correspondence with implementation fields, but the spec still refers to the specification field, which is conceptually distinct from the implementation field (even if it's public).

24. (10 points) List three distinct advantages of factory methods over constructors. (No more than 10 words each.)

    (a) ***It can return objects of any subtype, not just the declared type***

    (b) ***It can return null***

    (c) ***It can return existing objects, not just new ones. Examples include interning, singleton, coping, . . .***

    (d) It can be easily replaced by another factory, at run time or compile time

    (e) It can be used to intelligently choose which subtype to return

    (f) You can name it whatever you want

5 Code examples

25. (10 points) Write the output of running the following program's `main` method. (Hint: `IllegalArgumentException` and `NullPointerException` are subclasses of `RuntimeException`.)

```
public class TryCatchMystery {

    public static void main (String[] args) {
        try {
            method1();
            method2();
        } catch (IllegalArgumentException e) {
            System.out.println("main IllegalArgumentException");
        } catch (RuntimeException e) {
            System.out.println("main RuntimeException");
        }

    }

    public static void method1() {
        System.out.println("entered method1");
        try {
            method2();
        } catch (IllegalArgumentException e) {
            System.out.println("method1 IllegalArgumentException");
            throw new NullPointerException();
        } catch (NullPointerException e) {
            System.out.println("method1 NullPointerException");
            throw new NullPointerException();
        }
        System.out.println("exited method1");
    }

    public static void method2() {
        System.out.println("entered method2");
        throw new IllegalArgumentException();
    }

}
```

Answer:

```
entered method1
entered method2
method1 IllegalArgumentException
main RuntimeException
```

26. (8 points) In section, we looked at an example of the visitor pattern in which a PrintVisitor formatted and printed the text of a Book.

    ACME Publishing Company loves your PrintVisitor and has hired you to implement many custom formatting options. For example, one editor asked for fancy borders, so you wrote the following subclass:

    ```
    public class FancyPrintVisitor extends PrintVisitor { ...
    ```

    Another editor needed a large font size, so you wrote another subclass:

    ```
    public class LargePrintVisitor extends PrintVisitor { ...
    ```

    And so on, for semitic (right-to-left) printing and a host of other variants. Now, editors are asking to be able to combine arbitrarily many options at will, such as making some text both fancy and large. This is not convenient in your design.

    Explain what design technique would you use to solve this problem. Give the name of the technique and describe what you would have to modify or add to your code to implement the change.

    Use delegation. Instead of extending PrintVisitor, each special-feature printer would implement PrintVisitor and take an additional constructor that accepts another PrintVisitor. In visit() and its helper methods, each PrintVisitor would invoke the inner PrintVisitor. In this way, I could chain constructors and have all the print visitors do their print operation.

    Because this question was ambiguous as it was originally worded, we discarded it from the final score.

27. (20 points) ACME Publishing Company loves your PrintVisitor so much that they've asked to be able to use it to print magazines, too. The BasicPrintVisitor from section should print a magazine by writing the title of the magazine followed by the title and contents of each article to `System.out`. (Don't worry about whitespace formatting for now.)

    Implement the specified methods in the existing BasicPrintVisitor class and the new Magazine and Article classes below, using the visitor pattern. Note: in class, PrintVisitor appended text to a String that could later be printed to the console. For simplicity, just call `System.out.println` directly instead for the code you write here.

```
public class Magazine {
    private String title;
    private List<Article> articles;

    ...

    public String getTitle() {
        return title;
    }

    public List<Article> getArticles() {
        return Collections.unmodifiableList(articles);
    }

    public void accept(PrintVisitor v) {
        v.visit(this);
        for (Article a : articles)
            a.accept(v);
    }
}
```

(continued on next page)

```
public class Article {
    private String title;
    private String contents; // the text of the article

    ...

    public String getTitle() {
        return title;
    }

    public String getContents() {
        return contents;
    }

    public void accept(PrintVisitor v) {
        v.visit(this);


    }
}

public class BasicPrintVisitor extends PrintVisitor {

    ...

    // This helper method is called by visit(Text t);
    private void visitMagazine(Magazine m) {
        System.out.println(m.getTitle());


    }

    // This helper method is called by visit(Text t);
    private void visitArticle(Article a) {
        System.out.println(a.getTitle());
        System.out.println(a.getContents());

    }
 }
```