Guidelines

1. Short answer

(a) If you ask your web browser to display the web page at http://www.cs.umd.edu/users/pugh/index.html, what will be the first line of the request sent by the web browser? Feel free to mark any places where it might be browser dependent.

   Answer: GET /users/pugh/index.html HTTP/1.1, although it might be 1.0 rather than 1.1.

(b) For each of the following, indicate which of the following design patterns is most likely to be relevant or used:

   i. A graphical user interface Answer: Observer
   ii. Reading a log file Answer: Decorator
   iii. Looking for coding errors in a class file Answer: Visitor

(c) Briefly, what does a continuous build system do and when? (One sentence should suffice).

   Answer: Build and test a software artifact everytime a change is committed to the source code version repository (actually, it only checks every few minutes, so might not run for every change).

(d) Describe two situations in which you would use branching in a version control system.

   Answer:
   • To allow updates to be made to a released version of a project while work continues on the new version (e.g., after releasing 1.0 and starting work on 1.1, you might need to make a patch to fix a problem in 1.0; this would be done in the 1.0 branch).
   • An experimental branch that is expected to be moved to the HEAD development, but isn’t ready to be shared with all developers yet.

2. Bounded wildcards in generics

Assume you have a generic class Foo<E>. The Foo class defines a method process that takes a collection as an argument.

Assume that we also have

class A { ... }
class B extends A { ... }

Foo<A> f;
Collection c1;
Collection<Object> c2;
Collection<A> c3;
Collection<B> c4;
Collection<String> c5;
Below are some possible definitions for the argument of `process` in `Foo<E>`. For each one, report which `cn` variables could be passed to `f.process` with that definition.

(a) `void process(Collection c)`  **Answer:** c1, c2, c3, c4, c5  
(b) `void process(Collection<?> c)`  **Answer:** c1, c2, c3, c4, c5  
(c) `void process(Collection<? extends E> c)`  **Answer:** c1, c3, c4  
(d) `void process(Collection<? super E> c)`  **Answer:** c1, c2, c3  
(e) `void process(Collection<E> c)`  **Answer:** c1, c3  
(f) `void process(Collection<Object> c)`  **Answer:** c1, c2

3. Decorator.

The `Iterator` interface is given below:

```java
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Write/implement the following two `Iterator` decorators:

(a) **Non-removing iterator** This decorator is used to create a decorator that prevents `remove` from being invoked on the decorated `Iterator` by throwing an `UnsupportedOperationException` if a call to `remove` is made.

```java
public class NonRemovingIterator<E> implements Iterator<E> {
    final Iterator<E> i;
    public NonRemovingIterator(Iterator<E> i) {
        this.i = i;
    }
    public boolean hasNext() {
        return i.hasNext();
    }
    public E next() {
        return i.next();
    }
    public void remove() {
        throw new UnsupportedOperationException("remove not supported");
    }
}
```

(b) **Enhanced iterator** This decorator is used to support an additional capability: a method `E curr()` that returns the value most recently returned by `next()`. Calling `curr()` is idempotent: calling it multiple times has no impact. Calling `curr()` before `next()` has been called results in a `IllegalStateException` being thrown.
public class EnhancedIterator<E> implements Iterator<E> {

    boolean canCallCurrent;
    E current;
    final Iterator<E> i;
    public EnhancedIterator(Iterator<E> i) {
        this.i = i;
    }
    public boolean hasNext() {
        canCallCurrent = false;
        return i.hasNext();
    }
    public E next() {
        current = i.next();
        canCallCurrent = true;
        return current;
    }
    public void remove() {
        i.remove();
        canCallCurrent = false;
    }
    public E curr() {
        if (!canCallCurrent) throw new IllegalStateException();
        return current;
    }
}

4. Concurrency

This question requires you to implement a variant of the BoundedBuffer example we discussed in class. The variant is that rather than putting or taking a single element at a time, a sequence of objects are added or remove at once, as an atomic operation. In other words, if one thread tries to add ("a", "b") and another thread tries to add ("c", "d"), then is result must not be that the queue contains ("a", "c", "b", "d").

We are asking you to provide two implementations: the first one is a correct implementation. The second one should be correct except that the operations might not be performed atomically (e.g., if asked to put 5 elements into the queue, and the queue can only hold 3 more elements, 3 elements will be added immediately, and the put operation will wait/block until more room is available and the remaining 2 elements can be added to the queue.

(a) Correct implementation

    public class BoundedQueue {
        final LinkedList lst = new LinkedList();
        final int capacity;

        /** Create a buffer that can hold up to capacity elements */
        public BoundedQueue(int capacity) {
            this.capacity = capacity;
        }
    }
/** Remove N elements and return them, in order. If the queue doesn’t contain N elements, the operation must block until N elements are available. * The elements must be taken as an atomic operation. If the thread is interrupted while waiting, an InterruptedException exception is thrown * and the interrupted call makes no changes to the BoundedQueue. */
public synchronized List takeN(int count) throws InterruptedException {
    while (count > lst.size())
        wait();
    LinkedList result = new LinkedList();
    for(int i = 0; i < count; i++) result.addLast(lst.removeFirst());
    notifyAll();
    return result;
}

/** Add a list of elements to the queue. If the queue contain contain all of the added elements, the operation blocks until they can all be added in a single atomic update that doesn’t exceed the capacity of the queue. * If the thread is interrupted while waiting, an InterruptedException exception is thrown * and the interrupted call makes no changes to the BoundedQueue. */
public synchronized void putAll(List addThese) throws InterruptedException {
    while (lst.size() + addThese.size() > capacity)
        wait();
    for(Object e : addThese) lst.addLast(e);
    notifyAll();
}

(b) Non-atomic implementation (but otherwise thread safe and correct)

public class BoundedQueue {
    final LinkedList lst = new LinkedList();
    final int capacity;

    /** Create a buffer that can hold up to capacity elements */
    public BoundedQueue(int capacity) {
        this.capacity = capacity;
    }

    private waitIgnoringInterrupts() {
        try {
            wait();
        } catch (InterruptedException e) {} 
    }

    /** Remove N elements and return them, in order. * The operation is not atomic and interrupts are ignored */
    public synchronized List takeN(int count) throws InterruptedException {
        LinkedList result = new LinkedList();
        while (count > 0) {
            while (lst.isEmpty())
                waitIgnoringInterrupts();
            result.addLast(lst.removeFirst());
            count--;
            notifyAll();
        }
        return result;
    }
}
result.addLast(lst.removeFirst());
notifyAll();
count--;
}
return result;
}

/** Add a list of elements to the queue.
   * The operation is not atomic and interrupts are ignored
   */
public synchronized void putAll(List addThese) throws InterruptedException {
    while(!addThese.isEmpty()) {
        while (lst.size() == capacity)
            waitIgnoringInterrupts();
        lst.addLast(addThese.removeFirst());
        notifyAll();
    }
}

5. Visitor Pattern
Assume we have the following classes/interfaces that are used to represent abstract syntax trees (ASTs).

public interface Node {
    public E accept(Visitor<E> v);
}

public class IntegerConstant implements Node {
    public final int value;
    public IntegerConstant(int value) { this.value = value; }
    public E accept(Visitor<E> v) {
        return v.visit(this);
    }
}

public class Plus extends Node {
    public final Node lhs, rhs;
    public PlusBinaryOp(Node lhs, Node rhs) {
        this.lhs = lhs;
        this.rhs = rhs;
    }
    public E accept(Visitor<E> v) {
        return v.visit(this);
    }
}

public interface Visitor<E> {
    public E visit(IntegerConstant n);
    public E visit(Plus n);
}
public class Evaluate implements Visitor<Integer> {
    public Integer evaluate(Node n) {
        return n.accept(this);
    }
    public Integer visit(IntegerConstant n) {
        return n.value;
    }
    public Integer visit(Plus n) {
        return n.lhs.accept(this) + n.rhs.accept(this);
    }
}