How can a method be well tested when it's inputs can't be clearly identified? Consider this method in Java:

```java
/** Return a date object representing the start of the next minute from now */
public Date nextMinuteFromNow() {
    long nowAsMillis = System.currentTimeMillis();
    Date then = new Date(nowAsMillis + 60000);
    then.setSeconds(0);
    then.setMilliseconds(0);
    return then;
}
```

There are two barriers to effectively testing this method:
1. There is no easy way to test corner cases; you're at the mercy of the system clock to supply input conditions.
2. When `nextMinuteFromNow()` returns, the time has changed. This means the test will not be an assertion, it will be a guess, and may generate low-frequency, hard-to-reproduce failures... flakiness! Class loading and garbage collection pauses, for example, can influence this.

Is `System.currentTimeMillis()` starting to look a bit like a random number provider? That's because it is! The current time is yet another source of non-determinism; the results of `nextMinuteFromNow()` cannot be easily determined from its inputs. Fortunately, this is easy to solve: make the current time an input parameter which you can control.

```java
public Date minuteAfter(Date now) {
    Date then = new Date(now.getTime() + 60000);
    then.setSeconds(0);
    then.setMilliseconds(0);
    return then;
}
```

// Retain original functionality
@Deprecated public Date nextMinuteFromNow() {
    return minuteAfter(new Date(System.currentTimeMillis()));
}

Writing tests for `minuteAfter()` is a much easier task than writing tests for `nextMinuteFromNow()`:

```java
public void testMinuteAfter () {
    Date now = stringToDate("2012-12-22 11:59:59.999PM");
    Date then = minuteAfter(now);
    assertEquals("2012-12-23 12:00:00.000AM", dateToString(then));
}
```

This is just one way to solve this problem. Dependency Injection and mutable Singletons can also be used.