In-Class Testing Exercise
CSci 3081W
Spring 2016

Suppose you are working for a software development company, and are given some code that was written by someone else who recently left the company. You and a couple co-workers must test whether the code is correct. The code you will be testing is a class called `Polyline`, which relies on a related class `Point`. A `Polyline` object is basically a sequence of connected line segments, i.e., its primary member variable is a sequence of points.

Suppose you have access to the following.

- A short requirements document covering polyline-related requirements. (Example requirement: *The system shall be able to do the following geometric operations to any polyline: translate, rotate, and scale.*)
- A detailed design document for the `Polyline` and `Point` classes. (A simple UML diagram for these two classes is below.)
- C++ code (both `.h` and `.cpp` files) for the `Polyline` and `Point` classes.

This exercise consists of the following questions:

1. Suppose your boss asks you to create a test plan for the `Polyline` class: it is error-free? Does it correctly implement the requirements and design? Etc. If you had to write a half-page email to your boss describing the test plan what would you include? For example, what tasks would you include in the test plan? And how would you distribute the work among yourself and your two co-workers? (Don't actually write the email, but outline it or in some other way describe quickly what would be included.)

2. For each of the three sets of test cases below, determine what type of test coverage it provides for the `Polyline::cornerCut()` code on the next page. Specifically, state whether

   - Every *condition* in the code is executed at least once, regardless of whether the condition evaluates to true or false.
   - Every *statement* in the code is executed at least once.
   - Every *branch* in the code is followed at least once. Put another way, every condition in the code must evaluate to true at least once, and evaluate to false at least once. (However, if the condition is compound, then only the entire condition — not each part — needs to evaluate to true and to false.)

   Note there are three test sets below, some of which involve more than one polyline, with each polyline in turn consisting of a number of points.

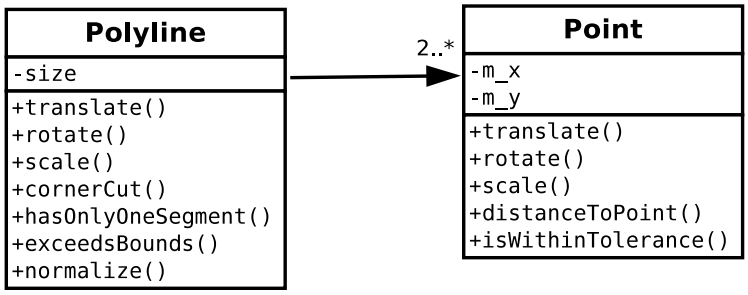
   (a) A single polyline with points $(0,0), (1,0), (2,0)$ and a minimum distance of .1.

   (b) A single polyline with points $(0,0), (.9,4), (1.0,4), (1.1,4), (2,2), (2,0)$ and a minimum distance of .2.

   (c) Two polylines. One with points $(0,0), (1,0), (1,1.1)$ and a minimum distance of .2; and the second with points $(0,0), (0.1,1), (2,0), (3,1)$ and a minimum distance of .2.

3. In Section 22.3, McConnell discusses "structured basis testing." What is the (minimal number of) test cases that structured basis testing would need for the `Polyline::cornerCut()` code? If none of the test sets in the previous question provided structured basis testing, then come up with a set of test cases that does. (Note: what constitutes a "test case" is not obvious here. As a challenge think about this; we'll discuss it in class as part of the exercise followup.)

Simple UML Diagram for Polyline and Point classes:



The polyline points are to be implemented as a member variable `m_points` that is of type `vector<Point>`.

Polyline::cornerCut() code

```
Polyline Polyline::cornerCut(double minDistance) const {
  double newX, newY;
  vector<Point> newPoints;

  // Always include first point of original polyline.
  newPoints.push_back(Point(m_points[0].getX(), m_points[0].getY()));
  // Check each interior point on original polyline.
  for (int i = 1; i < m_points.size()-1; i++){
    // If either leg at an interior point is too short, don't corner cut
    // --- just include that interior point in the new polyline.
    if (m_points[i-1].isWithinTolerance(pts[i], minDistance) ||
        m_points[i].isWithinTolerance(pts[i+1], minDistance)){
      newPoints.push_back(Point(m_points[i].getX(), m_points[i].getY()));
   }
    // Otherwise corner cut, and include, instead of the interior point,
    // the two points found by the corner cutting process.
    else {
      newX = 1.0/3.0 * m_points[i-1].getX() + 2.0/3.0 * m_points[i].getX();
      newY = 1.0/3.0 * m_points[i-1].getY() + 2.0/3.0 * m_points[i].getY();
      newPoints.push_back(Point(newX, newY));
      newX = 2.0/3.0 * m_points[i].getX() + 1.0/3.0 * m_points[i+1].getX();
      newY = 2.0/3.0 * m_points[i].getY() + 1.0/3.0 * m_points[i+1].getY();
      newPoints.push_back(Point(newX, newY));
    }
  }
  // Always include the last point of the original polyline.
  newPoints.push_back(Point(m_points[m_points.size()-1].getX(),
                            m_points[m_points.size()-1].getY()));
  // Create and return new polyline.
  return Polyline(newPoints);
}
```