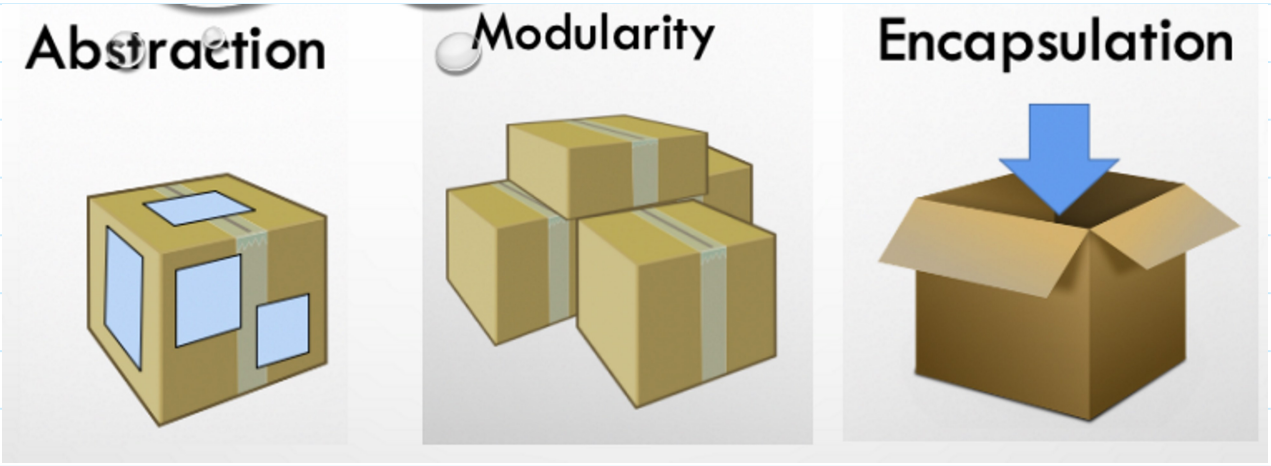
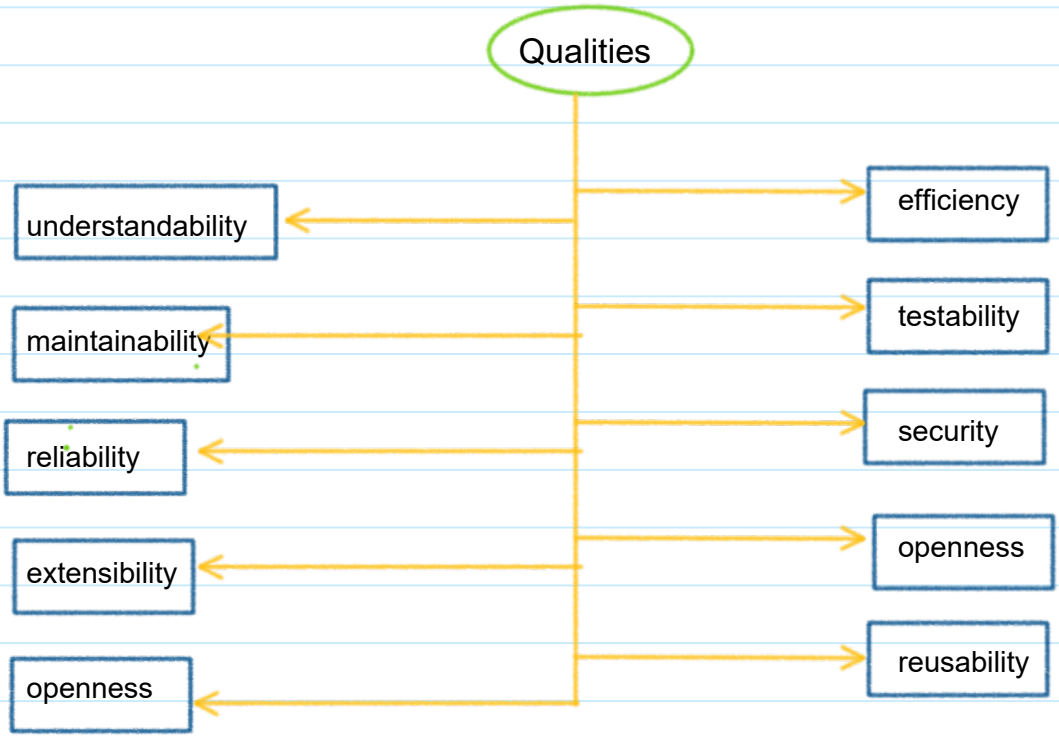
Software Engineering Principles

martes, 1 de febrero de 2022 21:41

DEFINITIONS



Software engineers aim to build quality products on time and within budget



CORE PROBLEM

- Are hard to teach.
- The proliferation and variation of principle definitions causes confusion among developers.
- Don't understand the core principles.
- Lack of unifying definitions hinders tools support.

BEST PRACTICES, PATTERNS, AND IDIOMS

- Patterns exemplify principles, by providing proven solutions to reoccurring problems in specific contexts.

- Idioms are techniques or solution for expressing a certain algorithm or data structure in a specific programming language, in a way that is consistency with certain principles.

Desirable Characteristics in The Software

- Reason about design decisions.
- Assess whether or how well a design either conforms to a principle.
- Balance choices between conflicting objectives and design alternatives.

OBSERVATIONS RELATIVE TO MODULARITY

- Modular reasoning – Good
- Extended modular reasoning – Okay
- Global reasoning – Bad

PARADIGM-INDEPENDENT DEFINITION FOR MODULARITY

• Localization of design decisions
• Low Coupling
• High Cohesion
• Modular Reasoning

```
// EXAMPLE N1 (Modularity)
1   // EXAMPLE N1 (Modularity)
2
3   public class Line {
4   private Point point1;
5   private Point point2;
6   public Line(Point point1, Point point2) {
7   this.point1 = point1;
8   this.point2 = point2;
9   }
10  public double ComputeLength() { /* .. */ }
11  }
12
13  public class Point {
14  private double x, y;
15  public Point(double x, double y) { this.x = x; this.y = y; }
16  public double getX() { return x; }
17  public void moveX(double deltaX) { x += deltaX;}
18  public double getY() { return y; }
19  public void moveY(double deltaY) { y += deltaY;}
20  }
```

OBSERVATIONS RELATIVE TO ABSTRACTION

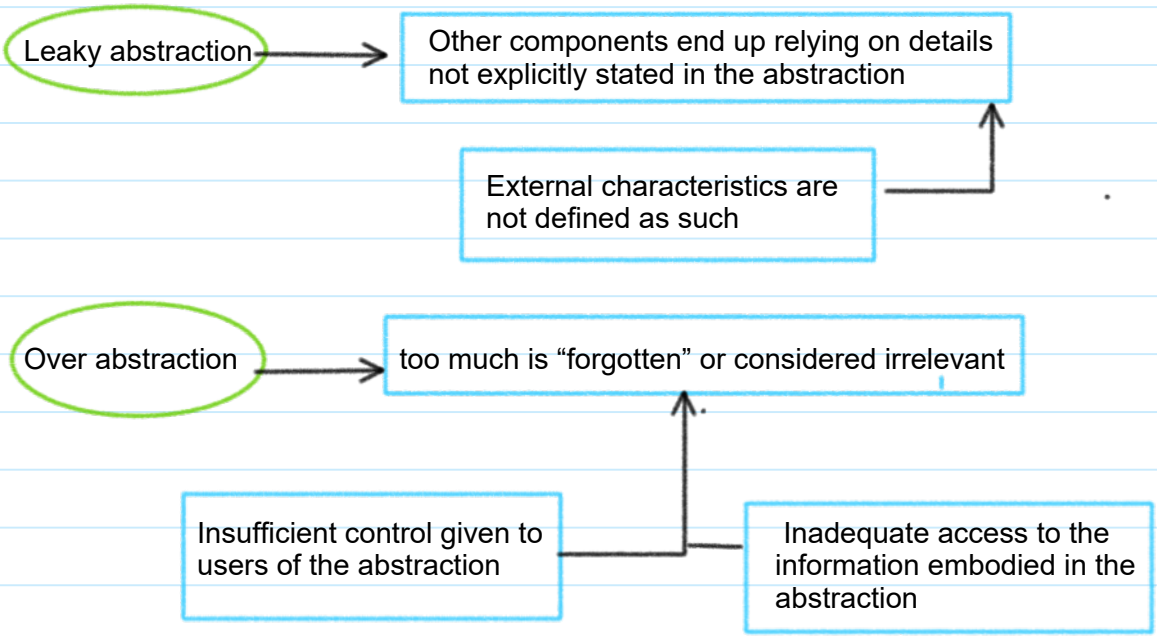
The abstraction is the act of bringing certain details to the forefront while suppressing all others.

• John Guttag said that “the essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context”

• From a software artifact perspective, an abstraction is anything that exposures certain details that others can use and rely on

• Some artifacts define their “public” details explicitly; others do not

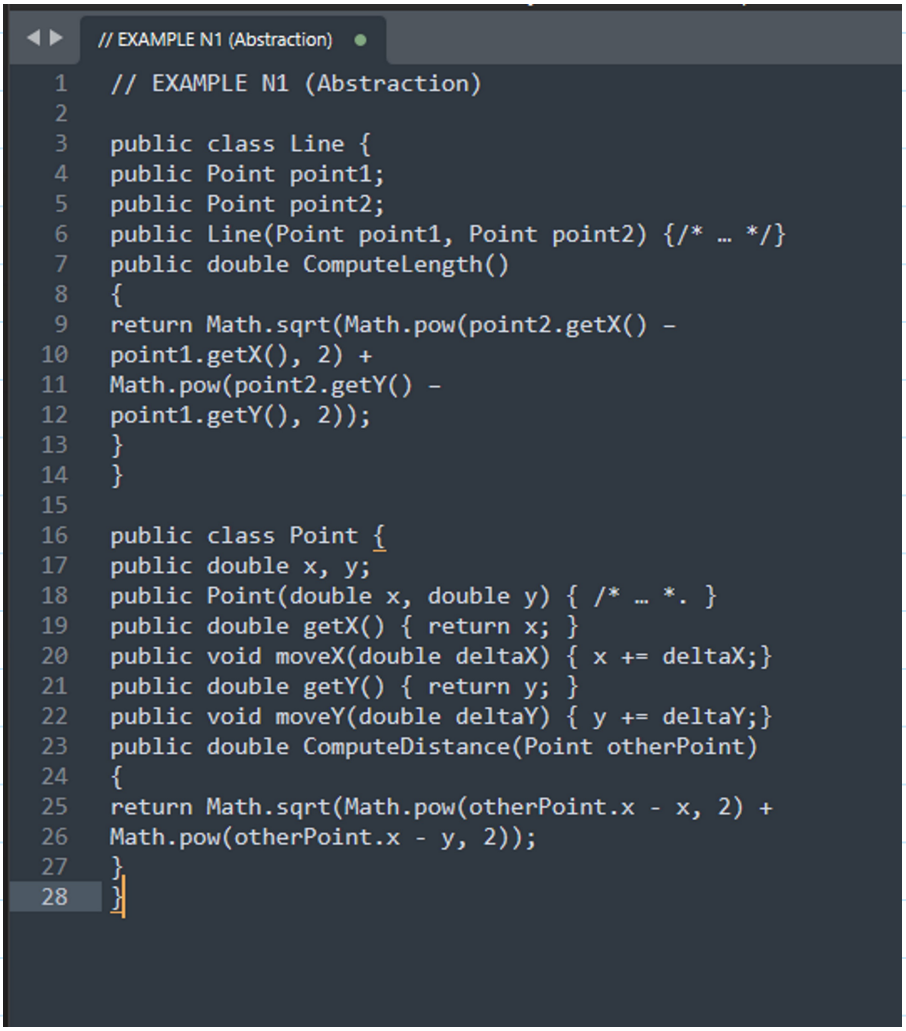
TWO COMMON PROBLEMS WITH ABSTRACTION



PARADIGM-INDEPENDENT DEFINITION FOR ABSTRACTION

Practices and Criteria:

• Meaningful labels and identifiers
• Context-aware labels and identifiers
• Abstraction completeness
• Abstraction sufficiency



OBSERVATIONS RELATIVE TO ENCAPSULATION

Although heavily used in OO, encapsulation is not unique to OO

Three categories of existing definition for encapsulation:

- The bundling of data with operations

  These definitions typically lack a proposition, rule, or practice that qualify them as principle definitions. Also, these definitions sometimes overlap modularity and its associated criteria.

- The hiding decisions behind logical barrier

  This category of definitions has given rise to access-restricting language constructs, such as the private and protected modifiers in class-based languages. Although definitions of this category are valuable, they do not capture encapsulation's full potential.

- The organization of components to minimize ripple effects

This category of definitions focuses on minimizing coupling – a modularization concern. By themselves, these definitions miss other important aspects of encapsulation and blur it with modularization.

PARADIGM-INDEPENDENT DEFINITION FOR ENCAPSULATION →

- Conceptual barriers
- Programmatic barriers
- Usage barriers

```
//Just Good Encapsulation
1  //Just Good Encapsulation
2
3  public class Line {
4  private Point point1;
5  private Point point2;
6  public Line(Point point1, Point point2) {/* … */}
7  public double Calc()
8  {
9  return Math.sqrt(Math.pow(point2.getX() –
10 point1.getX(), 2) +
11 Math.pow(point2.getY() –
12 point1.getY(), 2));
13 }
14 }
15
16 public class Point {
17 private double x, y;
18 public Point(double x, double y) { /* … *. }
19 public double getX() { return x; }
20 public void moveX(double deltaX) { x += deltaX;}
21 public double getY() { return y; }
22 public void moveY(double deltaY) { y += deltaY;}
23 public double ComputeDistance(Point otherPoint)
24 {
25 return Math.sqrt(Math.pow(otherPoint.x - x, 2) +
26 Math.pow(otherPoint.x - y, 2));
27 }
28 }
```