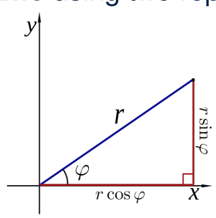
Uniform Access Principle

EECS3311 A: Software Design
Fall 2018

CHEN-WEI WANG

Uniform Access Principle (1)

- We may implement `Point` using two representation systems:



  - The *Cartesian system* stores the *absolute* positions of `x` and `y`.
  - The *Polar system* stores the *relative* position: the angle (in radian) `phi` and distance `r` from the origin (0.0).
- How the `Point` is implemented is irrelevant to users:
  - **Imp. 1**: Store `x` and `y`. [Compute `r` and `phi` on demand]
  - **Imp. 2**: Store `r` and `phi`. [Compute `x` and `y` on demand]
- As far as users of a `Point` object `p` is concerned, having a *uniform access* by always being able to call **`p.x`** and **`p.y`** is what matters, despite **Imp. 1** or **Imp. 2** being current strategy.

Uniform Access Principle (2)

```
class
  POINT
create
  make_cartisian, make_polar
feature -- Public, Uniform Access to x- and y-coordinates
  x : REAL
  y : REAL
end
```

- A class `Point` declares how users may access a point: either get its *x* coordinate or its *y* coordinate.
- We offer two possible ways to instantiating a 2-D point:
  - `make_cartisian (nx: REAL; ny: REAL)`
  - `make_polar (nr: REAL; np: REAL)`
- Features `x` and `y`, from the client's point of view, cannot tell whether it is implemented via:
  - ***Storage*** [`x` and `y` stored as real-valued ***attributes***]
  - ***Computation*** [`x` and `y` defined as ***queries*** returning real values]

Uniform Access Principle (3)

Let's say the supplier decides to adopt strategy **Imp. 1**.

```
class POINT -- Version 1
feature -- Attributes
  x : REAL
  y : REAL
feature -- Constructors
  make_cartisian(nx: REAL; nx: REAL)
    do
      x := nx
      y := ny
    end
end
```

- Attributes `x` and `y` represent the *Cartesian system*
- A client accesses a point `p` via **`p.x`** and **`p.y`**.
  - ***No Extra Computations***: just returning current values of `x` and `y`.
- However, it's harder to implement the other constructor: the body of `make_polar (nr: REAL; np: REAL)` has to compute and store `x` and `y` according to the inputs `nr` and `np`.

Uniform Access Principle (4)

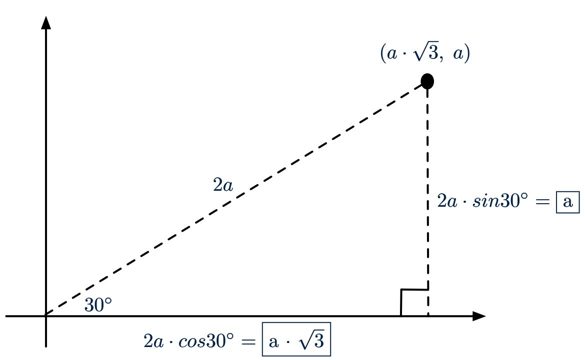
Let's say the supplier decides (*secretly*) to adopt strategy **Imp. 2**.

```
class POINT -- Version 2
feature -- Attributes
  r : REAL
  p : REAL
feature -- Constructors
  make_polar(nr: REAL; np: REAL)
    do
      r := nr
      p := np
    end
feature -- Queries
  x : REAL do Result := r × cos(p) end
  y : REAL do Result := r × sin(p) end
end
```

- Attributes `r` and `p` represent the *Polar system*
- A client **still** accesses a point `p` via **`p.x`** and **`p.y`**.
  - ***Extra Computations***: computing `x` and `y` according to the current values of `r` and `p`.

Uniform Access Principle (5.1)

Let's consider the following scenario as an example:



Note: $360° = 2\pi$

Uniform Access Principle (5.2)

```
1   test_points: BOOLEAN
2     local
3       A, X, Y: REAL
4       p1, p2: POINT
5     do
6       comment("test: two systems of points")
7       A := 5; X := A × √3; Y := A
8       create {POINT} p1.make_cartisian (X, Y)
9       create {POINT} p2.make_polar (2 × A, (1/6)π)
10      Result := p1.x = p2.x and p1.y = p2.y
11    end
```

- If strategy **Imp. 1** is adopted:
  - **L8** is computationally cheaper than **L9**. [`x` and `y` attributes]
  - **L10** requires no computations to access `x` and `y`.

  If strategy **Imp. 2** is adopted:
  - **L9** is computationally cheaper than **L8**. [`r` and `p` attributes]
  - **L10** requires computations to access `x` and `y`.

UAP in Java: Interface (1)

```java
interface Point {
      double getX();
      double getY();
}
```

- An interface `Point` defines how users may access a point: either get its *x* coordinate or its *y* coordinate.
- Methods `getX()` and `getY()` have no implementations, but *signatures* only.
- ∴ `Point` cannot be used as a *dynamic type*
- Writing `new Point(...)` is forbidden!

UAP in Java: Interface (2)

```
public class CartesianPoint implements Point {
  private double x;
  private double y;
  public CartesianPoint(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double getX() { return x; }
  public double getY() { return y; }
}
```

- `CartesianPoint` is a possible implementation of `Point`.
- Attributes `x` and `y` declared according to the *Cartesian system*
- `CartesianPoint` can be used as a *dynamic type*
  - `Point p = new CartesianPoint(3, 4)` allowed!
  - `p.getX()` and `p.getY()` return storage values

UAP in Java: Interface (3)

```
public class PolarPoint implements Point {
  private double phi;
  private double r;
  public PolarPoint(double r, double phi) {
    this.r = r;
    this.phi = phi;
  }
  public double getX() { return Math.cos(phi) * r; }
  public double getY() { return Math.sin(phi) * r; }
}
```

- `PolarPoint` is a possible implementation of `Point`.
- Attributes `phi` and `r` declared according to the *Polar system*
- `PolarPoint` can be used as a *dynamic type*
  - `Point p = new PolarPoint(3,` $\frac{\pi}{6}$`)` allowed! [$360° = 2\pi$]
  - `p.getX()` and `p.getY()` return computation results

UAP in Java: Interface (4)

```
1  @Test
2  public void testPoints() {
3    double A = 5;
4    double X = A * Math.sqrt(3);
5    double Y = A;
6    Point p1 = new CartisianPoint(X, Y); /* polymorphism */
7    Point p2 = new PolarPoint(2 * A, Math.toRadians(30)); /* polymorphism
8    assertEquals(p1.getX(), p2.getX());
9    assertEquals(p1.getY(), p2.getY());
10 }
```

How does *dynamic binding* work in **L9** and **L10**?

- `p1.getX()` and `p1.getY()` return storage values
- `p2.getX()` and `p2.getY()` return computation results

Uniform Access Principle (6)

The *Uniform Access Principle*:

- Allows clients to use services (e.g., `p.x` and `p.y`) regardless of how they are implemented.
- Gives suppliers complete freedom as to how to implement the services (e.g., Cartesian vs. Polar).
  - No right or wrong implementation; it depends!

access \ calculation	efficient	inefficient
frequent	COMPUTATION	STORAGE
infrequent	STORAGE if "convenient" to keep its value up to date COMPUTATION otherwise	

- Whether it's storage or computation, you can always change *secretly*, since the clients' access to the services is *uniform*.

Index (1)