Evaluating a Class Design

Evaluation is needed to accept, revise or reject a class design.

Five aspects to be evaluated:

- Abstraction: does it provide a useful one?
- Responsibilities: are they reasonable for the type?
- Interface: is it clean, simple?
- Usage: do we provide the “right” set of methods?
- Implementation: reasonable?
Tests for Adequacy of Abstraction

Identity:
Are class purpose and method purposes well-defined and connected?

Clarity:
Can purpose of class be given in brief, dictionary-style definition?

Uniformity:
Do operations have uniform level of abstraction?
class Date:
    Date represents a specific instant in time, with millisecond precision.

class TimeZone:
    TimeZone represents a time zone offset, and also figures out daylight savings.
Evaluating Class Design

Tests for Adequacy of Responsibilities

Clear:
Does class have specific responsibilities?

Limited:
Do responsibilities fit the abstraction (no more/less)?

Coherent:
Do responsibilities make sense as a whole?

Complete:
Does class completely capture the abstraction?

class Complex {
private:
    double Real, Imag;
public:
    Complex(double R = 0.0, double I = 0.0);
    double getReal() const;
    double getImag() const;
    void setReal();
    void setImag();
    double Magnitude() const;
};
## Tests for Adequacy of Interface

### Naming:
- Do names clearly express the intended effect?

### Symmetry:
- Are names and effects of pairs of inverse operations clear?

### Flexibility:
- Are methods adequately overloaded?

### Convenience:
- Are default values used when possible?
Example of Poor Naming

```cpp
class ItemList {
private:
    // ...
public:
    void Delete(Item item);
    // Take Item’s node out of list and delete Item

    void Remove(Item item);
    // Take Item’s node out of the list but do not delete Item

    void Erase(Item item);
    // Keep Item’s node in List, but with no information
};
```

Hard to remember difference!
Tests for Adequacy of Usage

Examine how objects of the class are used in different contexts (see below…)

Incorporate all operations that may be useful in these contexts… up to a point…

```cpp
class Location {
private:
    int xCoord, yCoord;  //coordinates
public:
    Location(int x = 0, int y = 0);
    int xCoord();        //return xCoord value
    int yCoord();        //return yCoord value
};

// usage:
Location point(100,100);
// shift point:
point = Location(point.xCoord()+5, point.yCoord()+10);
```

It's so complex!
class Location {
    private:
        int xCoord, yCoord;  //coordinates
    public:
        Location(int x = 0, int y = 0);
        int XCoord();    //return xCoord value
        int YCoord();    //return yCoord value
        void ShiftBy(int dx, int dy);  // shift by relative coordinates
    }

    // Revised usage:
    Location point(100,100);

    point. ShiftBy(5, 10);  // shift point
Implementation

Evaluating Class Design

Least important, mostly easily changed aspect to be evaluated.
- poorly engineered designs lead to problematic implementations
- massaging a problematic implementation (without redesign) rarely produces any effective improvement
- it’s only code… the issues here are primarily language syntax and semantics

Overly complex implementation may mean:
- class is not well conceived
- class has been given too much responsibility