

Information Hiding

- Do not expose internal information of a module unless necessary
 - E.g., private fields, getter & setter methods

N. Meng, B. Ryder

8

8

Abstraction & Refinement

- Abstraction
 - To manage the complexity of software,
 - To anticipate detail variations and future changes
- Refinement
 - A top-down design strategy to reveal low-level details from high-level abstraction as design progresses

N. Meng, B. Ryder

9

9

Abstraction to Reduce Complexity

- We abstract complexity at different levels
 - At the highest level, a solution is stated in broad terms, such as “process sale”
 - At any lower level, a more detailed description of the solution is provided, such as the internal algorithm of the function and data structure

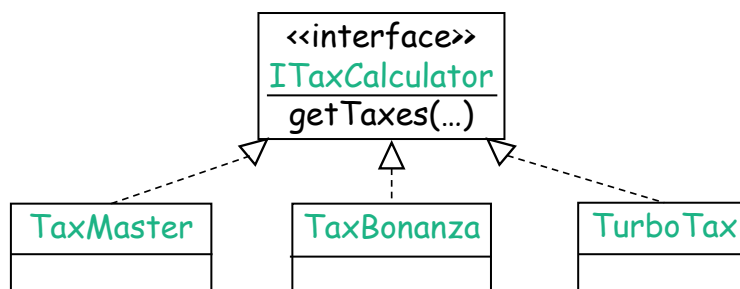
N. Meng, B. Ryder

10

10

Abstraction to Anticipate Changes

- Define interfaces to leave implementation details undecided
- Polymorphism



N. Meng, B. Ryder

11

11

Refinement

- The process to reveal lower-level details
 - High-level architecture software design
 - Low-level software design
 - Classes & objects
 - Algorithms
 - Data

N. Meng, B. Ryder

12

12

Refactoring

"...the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure"
--Martin Fowler

- Goal: to make software easier to integrate, test, and maintain.

N. Meng, B. Ryder

13

13

S.O.L.I.D Principles of OOD

Robert Martin

- S - Single-responsibility principle
- O - Open-closed principle
- L - Liskov substitution principle
- I - Interface segregation principle
- D - Dependency Inversion Principle

N. Meng, B. Ryder

14

14

A Running Example

```
class Circle {
    public float radius;

    public Circle(float radius) {
        this.radius = radius;
    }
}

class Square {
    public float length;

    public Square(float length) {
        this.length = length;
    }
}
```

N. Meng, B. Ryder

15

15

Single-responsibility principle

Robert Martin

- A class should have only one job.
 - Modularity, high cohesion, low coupling
- Sum up the areas for a list of shapes?

```
class AreaCalculator {
    protected List<Object> shapes;
    public AreaCalculator (List<Object> shapes) {
        this.shapes = shapes;
    }
    public float sumArea() {
        // logic to sum up area of each shape
    }
}
```

N. Meng, B. Ryder

16

16

O - Open-closed principle

- Objects or entities should be open for extension, but closed for modification.
- Add a new kind of shape, such as Triangle?

```
interface Shape {
    public float area();
}
class Triangle implements Shape { ... }
...
class AreaCalculator {
    protected List<Shape> shapes;
    public float sumArea() {
        float sum = 0;
        for (Shape s : shapes) {    sum += s.area(); }
        ...
    } ...
}
```

N. Meng, B. Ryder

17

17

L - Liskov substitution principle

- Let $q(x)$ be a property provable about objects of x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .
- Every subclass/derived class should be substitutable for their base/parent class.

```
class Triangle implements Shape {
    ...
    public float area () { return -1;}
}
```

N. Meng, B. Ryder

18

18

I - Interface segregation principle

- A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
- Interface design

```
interface Shape{
    ...
    public int numEdges();
}
```

N. Meng, B. Ryder

19

19

D - Dependency Inversion principle

- Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

```
class AreaCalculator{
    protected float radius;
    protected float length;
    public AreaCalculator(...,
        float param) {
        ...
        if (...//is a square)
            this.length = param;
        else // is a circle
            this.radius = param;
    }
}
```

X

N. Meng, B. Ryder

20

20

Software Design Practices Include:

- Two stages
 - High-level: Architecture design
 - Define major components and their relationship
 - Low-level: Detailed design
 - Decide classes, interfaces, and implementation algorithms for each component

N. Meng, B. Ryder

21

21

How to Do Software Design?

- Reuse or modify existing design models
 - High-level: Architectural styles
 - Low-level: Design patterns, Refactorings
- Iterative and evolutionary design
 - Package diagram
 - Detailed class diagram
 - Detailed sequence diagram