System Analysis and Design

L12. Design Engineering

Overview

- What is design engineering?
- How to do software design?
- Principles, concepts and practices

Design Engineering

- The process of making decisions about HOW to implement software solutions to meet requirements
- Encompasses the set of concepts, principles, and practices that lead to the development of high-quality systems

包括设计的概念，原理和实践.

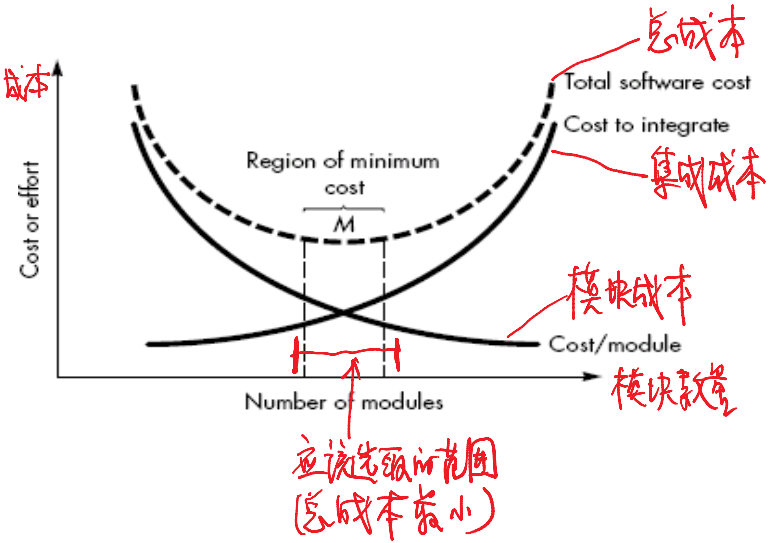
Concepts in Software Design

- Modularity
- Cohesion & Coupling
- Information Hiding
- Abstraction & Refinement
- Refactoring

Modularity

- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements
- Divide-and-conquer

Modularity and Software Cost



内聚与耦合
Cohesion & Coupling

- Cohesion
  - The degree to which the elements of a module belong together 模块内元素之间相关联的程度.
  - A cohesive module performs a single task requiring little interaction with other modules
- Coupling
  - The degree of interdependence between modules 模块间之相依之的程度
- High cohesion and low coupling ← 设计原则

Information Hiding

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

Abstraction & Refinement

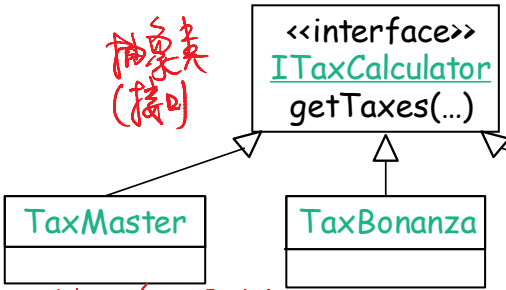
- Abstraction 抽象(概念化,简洁化,精炼)
  - To manage the complexity of software, 管控复杂度
  - To anticipate detail variations and future changes 事先考虑可能的选择与将来的变化
- Refinement 细化,精化
  - A top-down design strategy to reveal low-level details from high-level abstraction as design progresses 自上而下的策略,从高层抽象开始逐步揭示低层的细节.

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

Abstraction to Anticipate Changes

- Define interfaces to leave implementation details undecided
- Polymorphism



Refinement 细化

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

从高层的整体结构细化为低层的类和方法等

Refactoring

重构

"…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

不改变外部行为 } 改变软件
提高内部结构

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

使软件易于集成、测试、维护等

OOD的原则

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

Robert Martin

- S – Single-responsibility principle 单一职责原则
- O – Open-closed principle 开放封闭原则
- L – Liskov substitution principle Liskov替换原则
- I – Interface segregation principle 接口隔离原则
- D – Dependency Inversion Principle 依赖倒置原则

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;
}
}
```

圆（半径）

矩形（边长）

S – 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
  }
}
```

只做面积求和计算
也算一种工作.

O - Open-closed principle

对扩展开放，对修改封闭

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

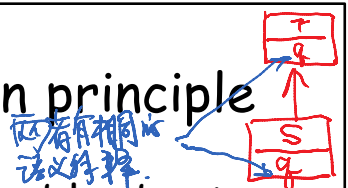
```
interface Shape {  public float area();
}
class Triangle implements Shape { … }
…
class AreaCalculator {
protected List<Shape> shapes;
public float sumArea() {
   float sum = 0;
   sum += s.area(); }
} …
```

实现三角形

没有改变

允许扩展其它形状的类型，而不允许修改接口

L - Liskov substitution principle



应该有相同的语义解释.

- 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. 所有子类型必须兼容父类型
- Every subclass/derived class should be substitutable for their base/parent class. 在继承关系中,父类定义的属性和方法若在子类中使用,应该具有相同语义解释.

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

X

与父类的要求不一致(不同的解释)

I - Interface segregation principle

接口隔离原则

- 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

用户无用的方法应该从接口中隔离出去.

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

边数

对用户无用！用类似面积的计算.

依赖倒置原则

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 Connection con;
      public AreaCalculator(…, MySQLConnection con){
          …
          this.con = con;
      }
  }
```

Abstraction

X

Concrete 直接依赖于 Connection.

对象可能依赖于低层的具体实现，而应该依赖于高层抽象.

Software Design Practices

- 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

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