Software Application Design Concepts and Principles

Learn and apply key Principles that facilitate Repeatable and Quality Software Designs

Boniface Kabaso

March 5, 2016

Outline

Outline

Outline

Software Design

Design Realities

1. Design, by nature, is a series of trade-offs.
2. Every choice has a good and bad side.
3. You make your choice in the context of overall criteria defined by necessity.
4. Good and bad are not absolutes.
5. A good decision in one context might be bad in another.
6. If you don't understand both sides of an issue, you cannot make an intelligent choice.
7. If you don't understand all the ramifications of your actions, you're not designing at all.

Outline

Software Design

Bad Software Design

Rigidity

Single change causes a cascade of subsequent changes in dependent modules

Fragility

Tendency of a Program to break in many places when a Single change is made.

Immobility

Refactoring is Hard and Difficult

Software Design

Bad Software Design

Viscosity

Difficult to do the right thing

Needless Complexity

Overdesign, too many unnecessary features

Needless Repetition

Mouse abuse (Cut and paste addiction)

Opacity

Code does indeed age

Outline

Software Design
Coding House Rules

Variables
- Camel case with lower case as first letter
- Use Intention Revealing and Pronounceable Names

Methods or Functions
- Camel case with lower case as first letter
- Use verb names and Limit Method Parameters to 3
- Method length should hardly ever be 15 lines long

Classes, Interfaces, Abstract and Objects
- Camel Case with Upper case for First Letter
- Use Noun Names and House them in a Package

Software Design
Coding House Rules

Packages
- Package names should be in lower case
- Use for grouping around themes

Modules
- Modules names should be lower case
- Belong to a project

Projects
- Project names should be lower case
- Project should be the top container for your jar or library

Outline

Outline

Object Oriented Languages

Features of OO languages

Any Language that claims to be Object Oriented must at least pass the test on these three Features

1. Encapsulation-to enforces modularity
2. Inheritance -to pass knowledge down
3. Polymorphism- to take any shape

Outline

Object Oriented Languages

Encapsulation

Listing 1: Code Sample for Encapsulation

```
1  public class CadreBean {
2      private String name;
3      public String getName() {
4          return name;
5      }
6      public void setName(String name) {
7          this.name = name;
8      }
9  }
```

Outline

Object Oriented Languages

Inheritance

Inheritance is the most abused OO principle. To get in right keep a clear definition of these

- Concrete class
- Interface class
- Abstract class

Object Oriented Languages

Inheritance

Listing 2: Code Sample for Encapsulation

```
1
2   interface A {}
3   interface B{}
4   abstract class C {}
5   class D {}
6
7   A extends B {}
8   D extends C {}
9   D implements A, B {}
```

Outline

Object Oriented Languages

Polymorphism

There are about four types of Polymorphism and we shall only deal with the last one

- Ad hoc Polymorphism
- Polytypism Polymorphism
- Parametric Polymorphism
- Subtyping Polymorphism

Object Oriented Languages

Subtyping Polymorphism

Listing 3: Subtyping Polymorphism Sample Code

```
1
2  public interface Vegetarian {}
3  public class Animal{}
4  Cow extends Animal implements ↩
       Vegetarian {}
5
6  Cow c = new Cow();
7  Animal a = c;
8  Vegetarian v = c;
9  Object o=c;
```

All the reference variables **c,a,v,o** refer to the same **Cow** object in the heap

Industry Day

Panel Discussion

Outline

Outline

Software Design
The Design Principles

We shall deal with six design Principles

- The Dependency-Inversion Principle (DIP)
- The Liskov Substitution Principle (LSP)
- The Open/Closed Principle (OCP)
- The Single-Responsibility Principle (SRP)
- The Interface Segregation Principle (ISP)
- The Principle of Least Knowledge (PLK)

Outline

The Design Principles

The Dependency-Inversion Principle (DIP)

DIP

Depend upon abstractions. Do not depend upon concretions

Key Points:

1. Two classes are tightly coupled if they are linked together and are dependent on each other
2. Tightly coupled classes can not work independent of each other
3. High level modules should not depend upon low level modules.
4. Both should depend upon abstractions Abstractions should not depend upon details.

Outline

The Design Principles

The Liskov Substitution Principle (LSP)

LSP

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

The Design Principles

The Liskov Substitution Principle (LSP)

Rules for LSP compliance when creating subclasses of existing classes to retain substitutability

KEY POINTS:

1. Contract Rules
   - Preconditions cannot be strengthened in a subtype.
   - Postconditions cannot be weakened in a subtype.
   - Invariants of the supertype must be preserved in a subtype.
2. Variance Rules
   - There must be contravariance of the method arguments in the subtype.
   - There must be covariance of the return types in the subtype.
   - No new exceptions can be thrown by the subtype.

The Design Principles

The Liskov Substitution Principle (LSP)

Listing 4: LSP Violation Code

```
1
2   class Bird {
3     public void fly(){}
4     public void eat(){}
5   }
6   class Dove extends Bird {}
7   class Ostrich extends Bird{
8     fly(){
9       throw new ↩
            UnsupportedOperationException();
10    }
11  }
```

The Design Principles

The Liskov Substitution Principle (LSP)

Listing 5: LSP Violation Code

```
1  public BirdTest{
2    public static void main(String[] args){
3      List<Bird> birdList = new ArrayList<Bird>();
4      birdList.add(new Bird());
5      birdList.add(new Dove());
6      birdList.add(new Ostrich());
7      letTheBirdsFly ( birdList );
8    }
9    static void letTheBirdsFly ( List<Bird> birdList ){
10     for ( Bird b : birdList ) {
```

Outline

The Design Principles
The Open/Closed Principle (OCP)

OCP

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification

Key Points:

1. Behavior of the module can be extended as the requirements of the application change, we can extend the module with new behaviors that satisfy those changes.
2. Extending the behavior of a module does not result in changes to the source, or binary, code of the module.

Outline

The Design Principles

The Single-Responsibility Principle (SRP)

SRP

A class or module should have one,and only one, reason to change.

Key Points:

1. we define a responsibility as a reason to change
2. If a class assumes more than one responsibility, that class will have more than one reason to change
3. more Class files != more Complexity

Outline

The Design Principles
The Interface Segregation Principle (ISP)

ISP

Many specific interfaces are better than a single, general interface

Key Points:

1. Classes whose interfaces are not cohesive have "fat" interfaces.
2. Any interface we define should be highly cohesive.
3. An interface should be responsible for allowing an object to assume a SINGLE ROLE

Outline

The Design Principles or Law of Demeter

The Principle of Least Knowledge (PLK)

PLK

For an operation O on a class C, only operations on the following objects should be called:itself, its parameters, objects it creates, or its contained instance objects.

Key Points:

1. Dont have deep knowledge of your neighbour's neighbour.
2. Don't call any methods on an object where the reference to that object is obtained by calling a method on another object
3. Do not to obtain a reference to your neighbour's neighbour, but instead to allow the your neighbour to forward the request

Outline

The Packaging Principles

The Packaging Principles

- Helps us split a large software system into Packages.
- As software Applications grow in size and complexity, they require some kind of high level organization.
- Classes are convenient unit for organizing small applications but too finely grained to be used as the sole organizational unit for large applications.
- Something "larger" than a class is needed to help organize large applications called a package,component or module.

The Packaging Principle

The Packaging Principle

1. The Principles of Component Cohesion-Granularity
   - The Reuse/Release Equivalence Principle (REP)
   - The Common Reuse Principle (CReP)
   - The Common Closure Principle (CCP)
2. Principles of Component Coupling-Stability
   - The Acyclic Dependencies Principle (ADP)
   - The Stable-Dependencies Principle (SDP)
   - The Stable-Abstractions Principle (SAP)

Outline

The Principles of Component Cohesion

The Reuse/Release Equivalence Principle (REP)

REP

- A package must be created with reusable classes
- Either all of the classes inside the package are reusable, or none of them are
- Classes must be of the same family

The Principles of Component Cohesion

The Common Reuse Principle (CReP))

CReP

- Classes that tend to be reused together belong in the same package together
- Classes that arent reused together should not be grouped together

The Principles of Component Cohesion

The Common Closure Principle (CCP)

CCP

- Packages should not have more than one reason to change.
- Classes that change together, belong together.
- When a change to one class may dictate changes to another class, its preferred that these two classes be placed in the same package

Outline

Principles of Component Coupling

Principles of Component Coupling-Stability

- The next three principles deal with the relationships between components.
- Again we will run into the tension between developability and logical design.
- The forces that impinge on the architecture of a component structure are technical, political, and volatile.

Principles of Component Coupling

The Acyclic Dependencies Principle (ADP)

ADP

- The dependencies between packages must form no cycles.
- Cycles among dependencies of the packages composing an application should almost always be avoided.
- Packages should form a Directed Acyclic Graph (DAG).

Principles of Component Coupling

The Stable-Dependencies Principle (SDP)

SDP

- Depend in the direction of stability.
- Packages with fewer incoming, and more outgoing dependencies, are less stable

$$I = \frac{ce}{ce + ca} \tag{1}$$

where

- **Ca (afferent couplings):** The number of classes outside this component that depend on classes within this component
- **Ce (efferent couplings):** The number of classes inside this component that depend on classes outside this

Principles of Component Coupling

The Stable-Abstractions Principle (SAP)

SAP

- Stable packages should be abstract packages.
- More stable packages, containing a higher number of abstract classes, or interfaces, should be heavily depended upon.
- Less stable packages, containing a higher number of concrete classes, should not be heavily depended upon.

Principles of Component Coupling

The Stable-Abstractions Principle (SAP)

$$A = \frac{Na}{Nc} \tag{2}$$

where

- **Na** is the number of abstract or interface classes in the component
- **Nc** is the number of concrete classes in the component
- **A** is a measure of the abstractness of a component

Questions ?

Boniface Kabaso

kabasoB AT cput DOT ac DOT za