Software Engineering
Design: What, Why, and How

Omkarendra Tiwari

Department of IT
IIIT Allahabad

March 3, 2023

Design

What and Why

- Organization of software system elements and their interaction/association/collaboration, to achieve
  - Reduced complexity in maintaining and evolving the system
  - Satisfying the non-functional requirements
  - Developing cost-effective solution

Design: Get it Right

Look for a suitable Pattern/Style

Monolithic

All components are packed and deployed together.

N-tier Architecture

Divide application into components and organize their interaction in layered fashion to hide/secure crucial functionalities/data.

Microkernel

An application with a core and extensible supporting functionalities.

Microservices

Identify various services an application is providing; package, deploy, and evolve each service independent of the other.

Client-Server

Server, a software designed to receive requests and provide services connects with other clients over the network.

Pipe-filter

An application designed to transform its input and feed forward to another filter for transformation.

Design: Not Quite Right

Code Smells/Anti-pattern

Code smells are design flaws that impact maintainability, evolvability, readability and other aspects of the code.

Large Class/God Class

A class with too many responsibilities.

Feature Envy

A method which is more interested in the data/members of other class.

Long Method/Brain Method

A method with more than one responsibility.

code clone

Duplicate logic/code present at multiple places.

Design : Make it right

Refactoring

External behavior preserving transformations

Extract Class

Divide the set of methods into two sets and extract one set along with fields into a new class.

Extract Method

Decompose the method into multiple smaller methods with cohesive method body.

Move Method

Move the envied method to a proper class.

Lets Design

Design at Method Level

```
    void FiboPrime() {
      int i, n, a, b, t;
A.    printf("Value of N:");
B.    scanf("%d",&n);
C.    a = 0;
D.    if (n ==1)
E.        printf("Nth Term:%d",a);
F.    else {
G.        b = 1;
H.        for (i=3;i <=n;i++){
I.           t = a + b;
J.           a = b;
K.           b = t;
        }
      }
L.    printf("Nth Term:%d",b);
M.    for (i=2; i<=b/2;i++){
N.       if (b%i == 0)
O.          break;
      }
P.    if (b<=1 || i <=b/2)
Q.       printf("Not Prime");
R.    else
S.       printf("Prime");
    } Fibonacci Prime Code
```

Design at Method Level

```
   void FiboPrime() {
      int i, n, a, b, t;
A.    printf("Value of N:");
B.    scanf("%d",&n);
C.    a = 0;
D.    if (n ==1)
E.        printf("Nth Term:%d",a);
F.    else {
G.        b = 1;
H.        for (i=3;i <=n;i++){
I.           t = a + b;
J.           a = b;
K.           b = t;
          }
      }
L.    printf("Nth Term:%d",b);
M.    for (i=2; i<=b/2;i++){
N.       if (b%i == 0)
O.          break;
      }
P.    if (b<=1 || i <=b/2)
Q.       printf("Not Prime");
R.    else
S.       printf("Prime");
   }  Fibonacci Prime Code
```

- Get Input : {A,B}
- Compute Nth Fibonacci Term : {C-K}
- Print the Fibonacci Term : {L}
- Has a divisor other that one and self : {M-O}
- Prime Checker : {P-S}

Design at Method Level

```
   int fibonacci() {
     int n, a, b, t;
A.   printf("Value of N:");
B.   scanf("%d",&n);
C.   a = 0;
D.   if (n ==1)
E.      printf("Nth Term:%d",a);
F.   else {
G.      b = 1;
H.      for (i=3;i <=n;i++){
I.         t = a + b;
J.         a = b;
K.         b = t;
        }
     }
L.   printf("Nth Term:%d",b);
X.   return(b);
   }

   void isPrime(int b) {
     int i;
M.   for (i=2; i<=b/2;i++){
N.     if (b%i == 0)
O.        break;
     }
P.   if (b<=1 || i <=b/2)
Q.     printf("Not Prime");
R.   else
S.     printf("Prime");
   }
```

Decomposition #1

```
   void fiboPrime() {
     int n;
A.   printf("Value of N:");
B.   scanf("%d",&n);
X.   isPrime(fibonacci(n));
   }
   int fibonacci(int n) {
     int a, b, t;
C.   a = 0;
D.   if (n ==1)
E.      printf("Nth Term:%d",a);
F.   else {
G.      b = 1;
H.      for (i=3;i <=n;i++){
I.         t = a + b;
J.         a = b;
K.         b = t;
        }
     }
L.   printf("Nth Term:%d",b);
X.   return(b);
   }

   void isPrime(int b) {
     int i;
M.   for (i=2; i<=b/2;i++){
N.     if (b%i == 0)
O.        break;
     }
P.   if (b<=1 || i <=b/2)
Q.     printf("Not Prime");
R.   else
S.     printf("Prime");
   }
```

Decomposition #2

Design at Class Level

Class version 1.0

```
public class Student {
   String Name;
   String EnrollmentNo;
   Date dob;
   int age;
   String address;
   String email;
   long int mobile;
   String semester;
   String department;
   String branch;
   String compulsory_courses;
   String elective_courses;
   String blood_grp;
   String father_name;
   String mother_name;
   long int bank_account;
   long int aadhar;
   float fee_due;
   String library_card;
   float gpa;
}
```

Design at Class Level

Class version 1.0

```
public class Student {
   String Name;
   String EnrollmentNo;
   Date dob;
   int age;
   String address;
   String email;
   long int mobile;
   String semester;
   String department;
   String branch;
   String compulsory_courses;
   String elective_courses;
   String blood_grp;
   String father_name;
   String mother_name;
   long int bank_account;
   long int aadhar;
   float fee_due;
   String library_card;
   float gpa;
}
```

Class version 1.1

```
public class Student {
   String EnrollmentNo;
   PersonalData personalDetails;
   AcademicData academicDetails;
   Responsibility responsibility;
   FinancialData financialDetails;
   Contact contact;
}
public class PersonalData {
   String Name;
   Date dob;
   String blood_grp;
   long int aadhar;
}
public class Contact {
   String address;
   String email;
   long int mobile;
   String father_name;
   String mother_name;
}
public class FinancialData {
   long int bank_account;
   float fee_due;
}
public class AcademicData {
   String department;
   String branch;
   String semester;
   String compulsory_courses;
   String elective_courses;
   String library_card;
   float gpa;
}
```

Design at Class Level

Class version 1.0

```
public class Student {
   String Name;
   String EnrollmentNo;
   Date dob;
   int age;
   String address;
   String email;
   long int mobile;
   String semester;
   String department;
   String branch;
   String compulsory_courses;
   String elective_courses;
   String blood_grp;
   String father_name;
   String mother_name;
   long int bank_account;
   long int aadhar;
   float fee_due;
   String library_card;
   float gpa;
}
```

Class version 1.1

```
public class Student {
   String EnrollmentNo;
   PersonalData personalDetails;
   AcademicData academicDetails;
   Responsibility responsibility;
   FinancialData financialDetails;
   Contact contact;
}
public class PersonalData {
   String Name;
   Date dob;
   String blood_grp;
   long int aadhar;
}
public class Contact {
   String address;
   String email;
   long int mobile;
   String father_name;
   String mother_name;
}
public class FinancialData {
   long int bank_account;
   float fee_due;
}
public class AcademicData {
   String department;
   String branch;
   String semester;
   String compulsory_courses;
   String elective_courses;
   String library_card;
   float gpa;
}
```

Class version 1.2

```
public class TeachingStaff {
   String EmpId;
   PersonalData personalDetails;
   AcademicData academicDetails;
   Responsibility responsibility;
   FinancialData financialDetails;
   Contact contact;
}
```

Class version 1.2

```
public class NonTeachingStaff {
   String EmpId;
   PersonalData personalDetails;
   AcademicData academicDetails;
   Responsibility responsibility;
   FinancialData financialDetails;
   Contact contact;
}
```

Design at Class Level : Inheritance

Class version 1.3

```
  public class Student extends
InstituteMember {
     String EnrollmentNo;
     AcademicData academicDetails;
  }
  public class PersonalData {
     String Name;
     Date dob;
     String blood_grp;
     long int aadhar;
  }
  public class Contact {
     Address address;
     String email;
     long int mobile;
     Parents parent;
  }
  public class FinancialData {
     long int bank_account;
     float fee_due;
  }
  public class AcademicData {
     String department;
     String branch;
     String semester;
     String compulsory_courses;
     String elective_courses;
     String library_card;
     float gpa;
  }
```

Design at Class Level : Inheritance

Class version 1.3

```
   public class Student extends
InstituteMember {
      String EnrollmentNo;
      AcademicData academicDetails;
   }
   public class PersonalData {
      String Name;
      Date dob;
      String blood_grp;
      long int aadhar;
   }
   public class Contact {
      Address address;
      String email;
      long int mobile;
      Parents parent;
   }
   public class FinancialData {
      long int bank_account;
      float fee_due;
   }
   public class AcademicData {
      String department;
      String branch;
      String semester;
      String compulsory_courses;
      String elective_courses;
      String library_card;
      float gpa;
   }
```

Class version 1.3

```
   public abstract class
InstituteMember {
      PersonalData personalDetails;
      Responsibility responsibility;
      FinancialData financialDetails;
      Contact contact;
   }
```

Class version 1.3

```
   public class TeachingStaff extends
InstituteMember{
      String EmpId;
      AcademicData academicDetails;
   }
```

Class version 1.3

```
   public class NonTeachingStaff
extends InstituteMember{
      String EmpId;
      AcademicData academicDetails;
   }
```

Class version 1.3

```
   public class Address {
      String HouseNo;
      String Street;
      String district;
      String state;
      String country;
      long int pin;
   }
```

Further Increments

- AcademicData (history, current, grade-generation, courses)
- Library
- Gymkhana (Sports, Swimming Pool, Gym, Cultural)
- Projects (Social or Financial)
- Department

Caution with Inheritance

Substitutability

- Use *Inheritance* only when substitutability is required
- A derived class can not demand more resources and can not offer fewer services
- A user of base class should be able to use the derived class without knowing the difference
- If the criteria does not meet use *composition/delegation* instead

Class Diagram: Communicating Design and Architecture

Shapes and association

- Class: A rectangle with three parts
- Association: A straight line connecting two elements
  - Aggregation: An arrow with a hollow diamond head
  - Composition: An arrow with a filled diamond head
- Inheritance: An arrow with a triangle head
- Access Specifiers: + (public), # (protected), -(private)

Design Principles

Basic Principles

- YGNI: You Aren't Gonna Need It
- DRY: Don't Repeat Yourself

SOLID Principles

- SRP: Single Responsibility Principle
- OCP: Open-close Principle–Open for Extension but closed for modification
- LSP: Liskov's Substitution Principle–Inheritance should be used only for substitutability
- ISP: Interface Segregation Principle
- DIP: Dependency Inversion Principle

Further Readings

Highly Recommended

- https://www.yegor256.com/2014/11/20/seven-virtues-of-good-object.html

- https://martinfowler.com/architecture/
- https://microservices.io/patterns/monolithic.html