Software Design and Analysis Introduction

Instructor: Mehroze Khan

What is Software Engineering?

- *Software engineering* is the process of **solving customers' problems** by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints.

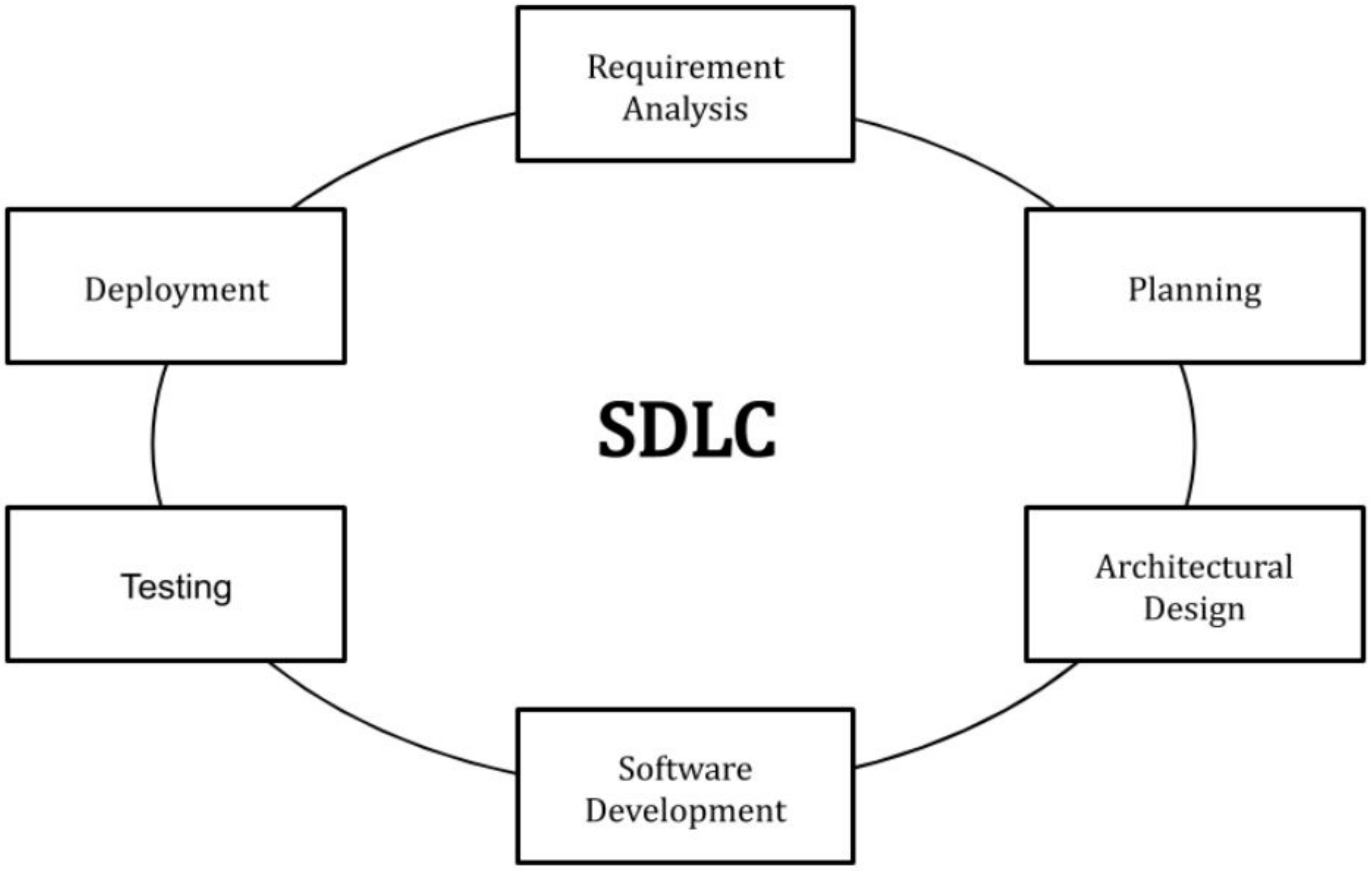
Stakeholders in Software Engineering

- **Users**. These are the people who will use the software. Their goals usually include doing enjoyable or interesting work and gaining recognition for the work they have done.
- **Customers** (also known as *clients*). These are the people who make the decisions about ordering and paying for the software. They may or may not be users – the users may work for them.
- **Software developers**. These are the people who develop and maintain the software, many of whom may be called software engineers.
- **Development managers**. These are the people who run the organization that is developing the software.

Software Development Lifecycle (SDLC)

- The Software Development Life Cycle (SDLC) refers to a methodology with clearly defined processes for creating high-quality software. SDLC methodology focuses on the following phases of software development:
  - Requirement analysis
  - Planning
  - Software design such as architectural design
  - Software development
  - Testing
  - Deployment/Maintenance

Software Development Lifecycle (SDLC)



OOP?

- Object-oriented programming is a method of implementation in which programs are organized as **cooperative collections of objects**, each of which represents an **instance of some class**, and whose classes are all members of a hierarchy of classes united via **inheritance relationships**.

Object Oriented Design

- Object-oriented design is a method of design encompassing the process of **object-oriented decomposition** and a notation for depicting both **logical** and **physical** as well as **static** and **dynamic** models of the system under design.

Object Oriented Analysis

- Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

Object Model

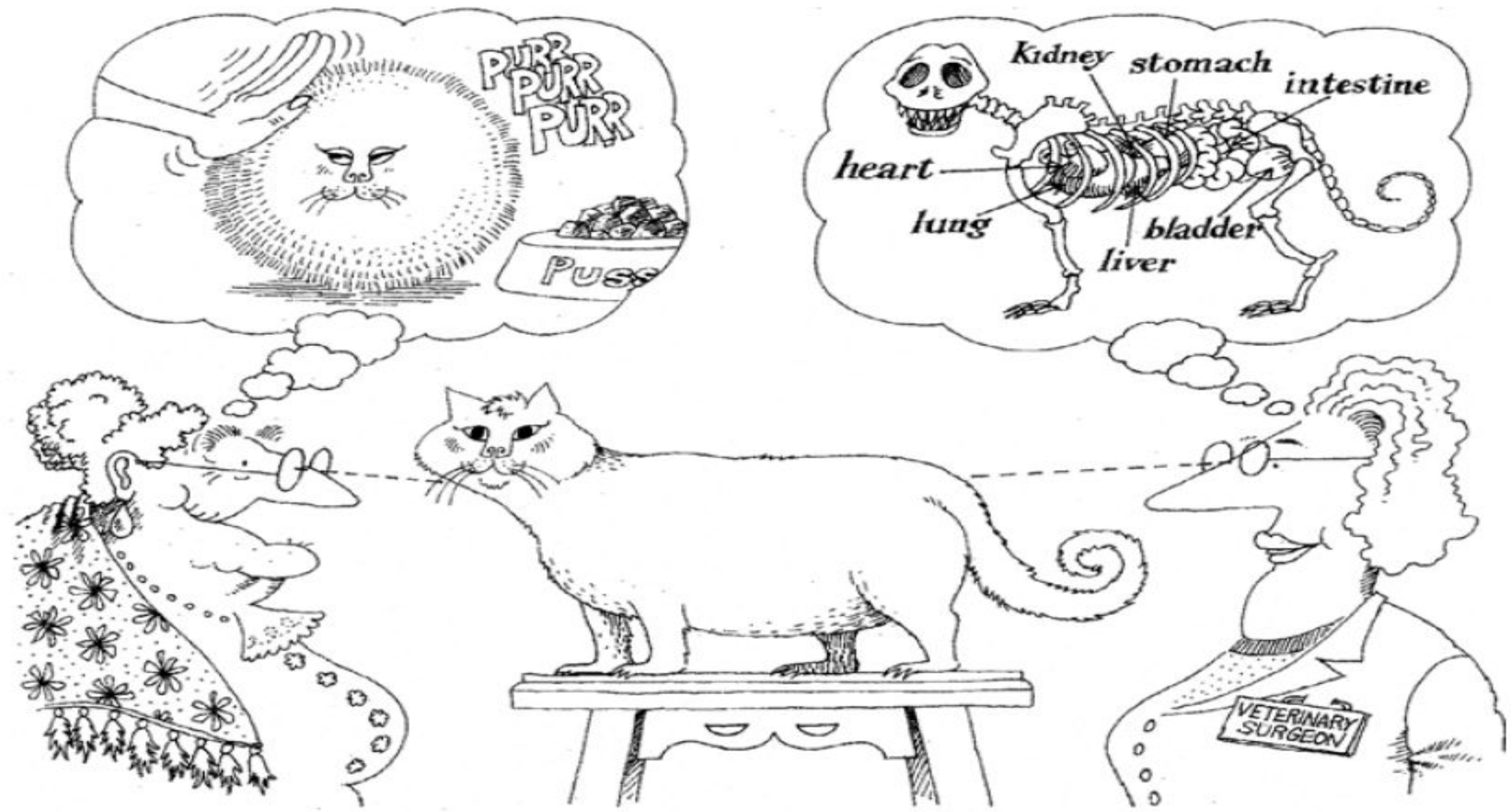
For all things object oriented, the conceptual framework is the object model. There are four major elements of this model:

1. Abstraction
2. Encapsulation
3. Modularity
4. Hierarchy

Abstraction

- An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
- An abstraction focuses on the outside view of an object and so serves to separate an object’s essential behavior from its implementation.

Abstraction



Abstraction focuses on the essential characteristics of some object, relative to the perspective of the viewer.

Abstraction

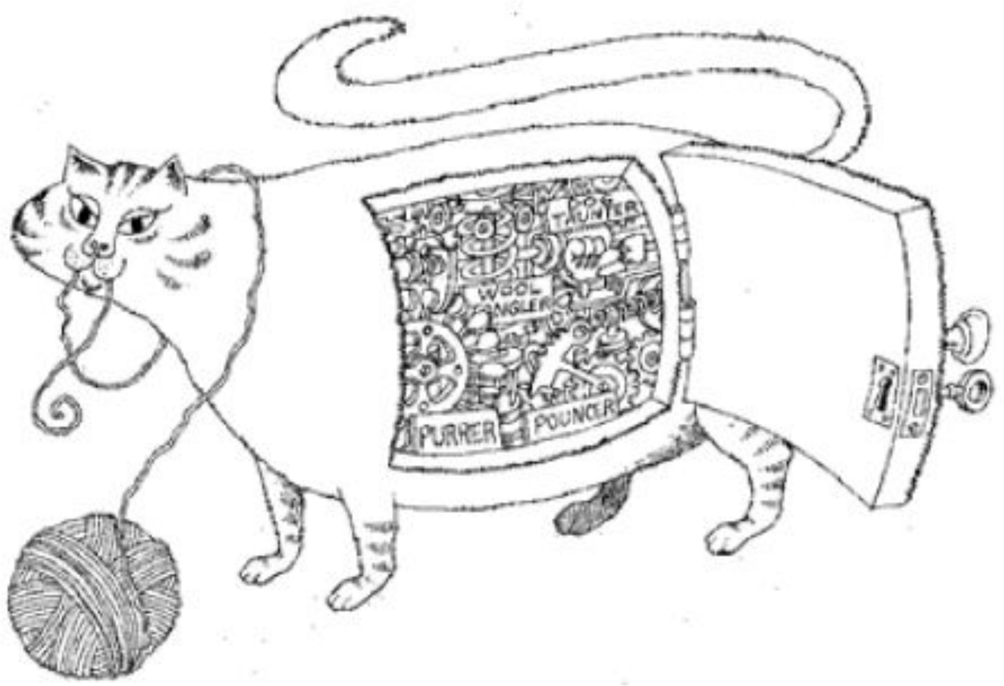
```
class Shape {
public:
    virtual void draw() = 0;        // Pure virtual function
};
// Derived class representing a circle
class Circle : public Shape {
public:
    void draw() {        // Implementation of draw for a circle
        cout << "Drawing a circle." << endl;
    }
};
// Derived class representing a rectangle
class Rectangle : public Shape {
public:
    void draw() {    // Implementation of draw for a rectangle
        cout << "Drawing a rectangle." << endl;
    }
};
```

```
int main() {
    Circle circle;              // Create a Circle object
    Rectangle rectangle;        // Create a Rectangle object
    circle.draw();              // Output: Drawing a circle.
    rectangle.draw();             // Output: Drawing a rectangle.
}
```

Encapsulation

- It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class.
- Encapsulation also involves restricting direct access to some of an object's components, which is a way to safeguard the internal state of the object from unintended interference and misuse.

Encapsulation



Encapsulation hides the details of the implementation of an object.

Encapsulation

```
class BankAccount {
private:
   string accountNumber;
   double balance; // Private attribute

public:
   BankAccount(string accNum, double initialBalance) {
        accountNumber = accNum;
        balance = initialBalance;
   }

   void deposit(double amount) {
      if (amount > 0) {
         balance += amount;
         cout << "$" << amount << " deposited. New balance: $" <<
balance << endl;
      }
   }
```

```
void withdraw(double amount) {
      if (amount > 0 && amount <= balance) {
         balance -= amount;
         cout << "$" << amount << " withdrawn. New balance: $" <<
balance << endl;
      } else {
         cout << "Insufficient balance" << endl;
      }
   }
   double getBalance() {
      return balance;
   }
};
int main() {
   BankAccount account("123456789", 500);
   account.deposit(200);
   account.withdraw(100);
   cout << "Account balance: $" << account.getBalance() << endl;
}
```

Modularity

- Partitioning a program creates a number of well-defined, documented boundaries within the program.
- Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules.
- The overall goal of the decomposition into modules is the reduction of software cost by allowing modules to be designed and revised independently

Modularity

```
// BankAccount.h
#ifndef BANKACCOUNT_H
#define BANKACCOUNT_H
#include <string>

class BankAccount {
private:
    string accountNumber;
    double balance;

public:
    BankAccount(string accNum, double initialBalance);
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance() const;
};
#endif // BANKACCOUNT_H
```

Modularity

```
// BankAccount.cpp
#include "BankAccount.h"
#include <iostream>

BankAccount::BankAccount(string accNum,
double initialBalance) {
    accountNumber = accNum;
    balance = initialBalance;
}

void BankAccount::deposit(double amount) {
    if (amount > 0) {
        balance += amount;
        cout << "$" << amount << " deposited. New
balance: $" << balance << endl;
    }
}
```

```
void BankAccount::withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        cout << "$" << amount << " withdrawn. New
balance: $" << balance << endl;
    } else {
        cout << "Insufficient balance" << endl;
    }
}

double BankAccount::getBalance() const {
    return balance;
}
```

Modularity

```
// main.cpp
#include <iostream>
#include "BankAccount.h"

int main() {
    BankAccount account("123456789", 500);
    account.deposit(200);
    account.withdraw(100);
    cout << "Account balance: $" << account.getBalance() << endl;
}
```

Hierarchy

- Hierarchy is a ranking or ordering of abstractions.
- The two most important hierarchies in a complex system are its class structure (the “is a” hierarchy) and its object structure (the “part of” hierarchy).
- **Examples of Hierarchy**
  - Single Inheritance (is a hierarchy)
  - Multiple Inheritance (is a hierarchy)
    - ❖ Clashes will occur when two or more super classes provide a field or operation with the same name or signature as a peer superclass.
    - ❖ Repeated inheritance occurs when two or more peer super classes share a common superclass. In such a situation, the inheritance lattice will be **diamond-shaped**, so the question arises, does the leaf class (i.e., subclass) have one copy or multiple copies of the structure of the shared superclass?
  - Aggregation (part of hierarchy)

Hierarchy (Single Inheritance)

```
// Base class
class Animal {
public:
   void eat() {
      cout << "This animal is eating." << endl;
   }
};
// Derived class
class Dog : public Animal {
public:
   void bark() {
      cout << "The dog is barking." << endl;
   }
};
int main() {
   Dog myDog;
   myDog.eat();
   myDog.bark();
}
```

Hierarchy (Aggregation)

```
// Part class
class Engine {
public:
   void start() {
      cout << "Engine started." << endl;
   }
};
// Whole class
class Car {
private:
   Engine engine;  // Engine is a part of Car

public:
   void start() {
      engine.start();  // Start the engine when the car starts
      cout << "Car started." << endl;
   }
};
```

```
int main() {
   Car myCar;

   // Car is composed of an Engine, so starting the car also
starts the engine
   myCar.start();  // Output: Engine started. Car started.

   return 0;
}
```

Diamond Problem

```
class A {
public:
    void display() {
      cout << "A's display()" << endl;
   }
};
class B : public A {
   // No override of display()
};
class C : public A {
   // No override of display()
};
```

```
class D : public B, public C {
   // No override of display()
};
int main() {
   D obj;
   obj.display();  // Which display()
should be called?
   return 0;
}
```

Here, D inherits display() twice from class A (via both B and C), causing ambiguity when calling obj.display(). The compiler will throw an error because it doesn't know whether to use B::A::display() or C::A::display().

soln:

```
D obj;
   obj.B::display(); // Explicitly calls A's display() via B
   obj.C::display(); // Explicitly calls A's display() via C
```

Solution in C++ (Virtual Inheritance)

```
class A {
public:
    virtual void display() {
        cout << "A's display()" << endl;
    }
};
class B : public virtual A {
    // No override of display()
};
class C : public virtual A {
    // No override of display()
};
```

```
class D : public B, public C {
    // No override of display()
};
int main() {
    D obj;
    obj.display();  // Which display() 
should be called?
    return 0;
}
```

When B and C inherit A virtually, only one instance of A exists in the inheritance hierarchy, even when D inherits from both B and C.
Hence, D can directly call display() without ambiguity.

Benefits of Object Model

- Use of the object model:
    - Helps us to exploit the expressive **power of object-based and object-oriented programming languages**
    - Encourages the **reuse** not only of software but of entire designs, leading to the creation of reusable application frameworks
    - Produces systems that are built on stable intermediate forms, which are more **resilient to change**
    - Appeals to the workings of **human cognition**

Object Oriented Methodology

The methodology has the following stages.

- **System conception**. Software development begins with business analysts or users conceiving an application and formulating tentative requirements.
- **Analysis**. The analyst scrutinizes and rigorously restates the requirements from system conception by constructing models. The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct.
  - The **analysis model** has two parts: the ***domain model***, a description of the real-world objects reflected within the system; and the ***application model***, a description of the parts of the application system itself that are visible to the user.
- **System design**. The development team devise a high-level strategy—the ***system architecture***— for solving the application problem.
- **Class design**. The class designer adds details to the analysis model in accordance with the system design strategy. The class designer elaborates both domain and application objects using the same OO concepts.
- **Implementation**. Implementers translate the classes and relationships developed during class design into a particular programming language, database, or hardware.

References

- Object-Oriented Analysis and Design with Applications, Grady Booch et al., 3rd Edition, Pearson, 2007.
- Timothy C. Lethbridge, Robert Laganaiere, Object-Oriented Software Engineering (2nd Edition), McGraw Hill, 2005
- Object-Oriented Modeling and Design with UML, Michael R. Blaha and James R. Rumbaugh, 2nd Edition, Pearson, 2005.