Dalhousie University
Faculty of Computer Science

CSCI 3132 – Object Orientation and Generic Programming

Week 3 – The Object Model

Most of slides and theoretical examples in this lecture have been taken or adapted from “Object-Oriented Analysis and Design with Applications” by Grady Booch et. al.

Contents for this week

- Elements of The Object Model
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
    - Inheritance
    - Aggregation
  - Typing
  - Concurrency
  - Persistence

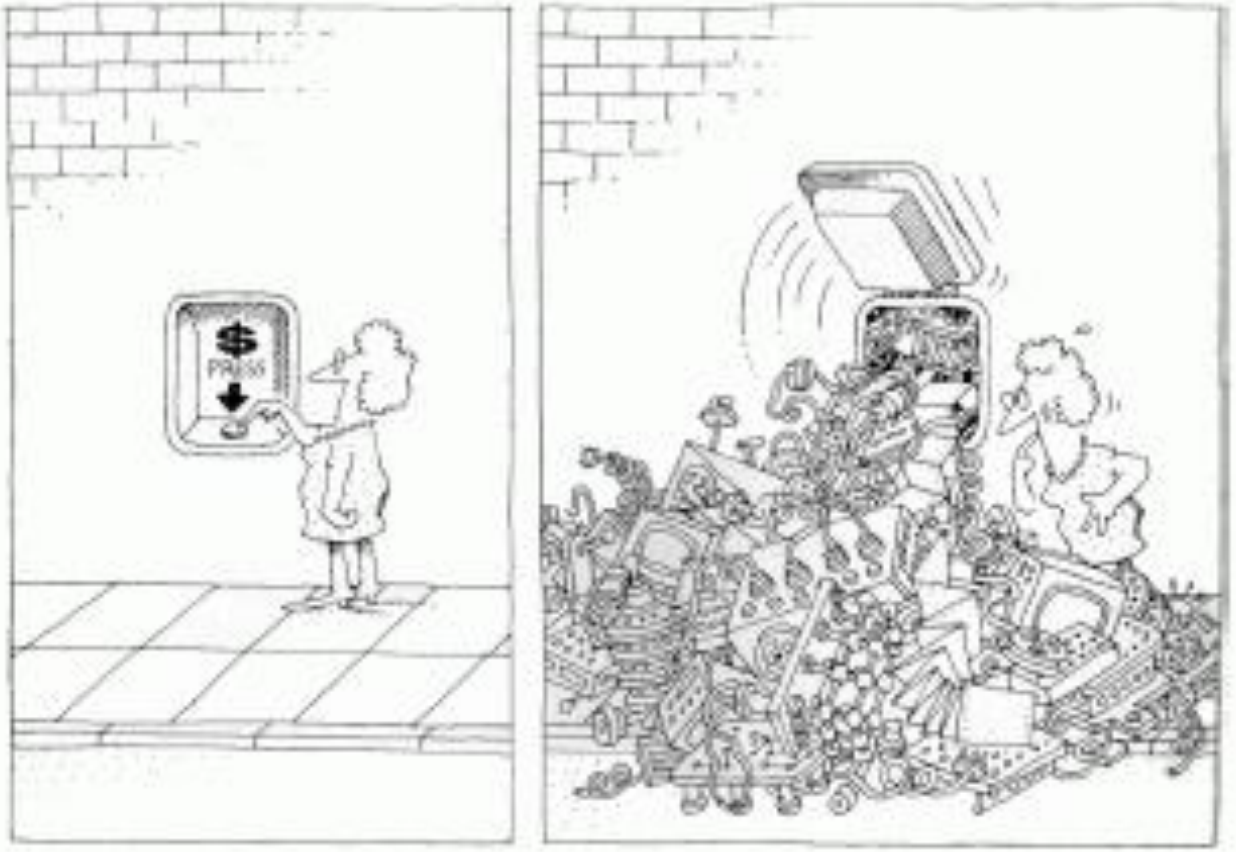
Motivation

- We focus on complex software
  - Applications exhibit a rich set of behaviors
    - Routing of air traffic on a busy airport
    - Software for self-driving vehicles
    - Google's back-end cluster

Complexity

- Where does complexity arise from?
  - Complexity of the problem domain
  - Difficulty of managing the development process
  - Flexibility possible through software
  - Characterizing the behavior of discrete systems

Managing Complexity



The task of the software development team
is to engineer the illusion of simplicity.

[Booch et al., p.9]

Managing Complexity

- Managing complex software systems through object-oriented paradigm
  - Analysis: Objects are the building blocks of software components
    - Object Oriented Decomposition
  - Design: software components enjoy a hierarchical structure
    - Is-A (or Kind-Of) and Part-Of" relations
  - Programming: use programming language supporting object-oriented concepts

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" *[Booch et al., p.42]*

- Building real-world models using object-oriented view of the world
- The object model encompasses the principles of object oriented programming

Object Oriented Design

"Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both a logical and physical, as well as static and dynamic models of the system under design" *[Booch et. al. p. 42]*

- Proper and effective structuring of a complex system
- Supports object-oriented decomposition
- Use of appropriate notation to express logical and physical design of a system

Object Oriented Programming

> " … 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 relationship … " *[Booch et al., p.41]*

- Proper and effective use of object-oriented language concepts
  - Objects, not algorithms, are the fundamental building blocks
  - Each object is an instance of some class
  - Classes may be related via inheritance (is-A) relationships

Types of Programming Languages

"Most programmers work in one language and use only one programming style. They program in a paradigm enforced by the language they use. Frequently, they have not been exposed to alternate ways of thinking about a problem, and hence have difficulty in seeing the advantage of choosing a style more appropriate to the problem at hand" *[Ch. 2, Ref. 40, Jenkins et. al.]*

- Several kinds of programming styles with the kind of abstractions they employ include:
  - Procedure-oriented → Algorithms
  - Object-oriented → Classes and objects
  - Logic-oriented → Goals (predicate Calculus)
  - Rule-oriented → If-Then rules
  - Constraint-oriented → Invariant relationships
- No single programming style that is best for all kinds of apps.

Object Oriented Languages

- A language is Object Oriented if:
  - It supports objects with an interface of named operations and a hidden local state (abstraction)
  - Objects have an associated type (class)
  - Types may inherit attributes from supertypes (classes from super- or base classes)
  - Smalltalk, C++, Object Pascal, Eiffel, CLOS, C# and Java are all OO languages
- A language is object based if it does not provide direct support for inheritance
  - Ada83 is object based (Ada95 supports OO)

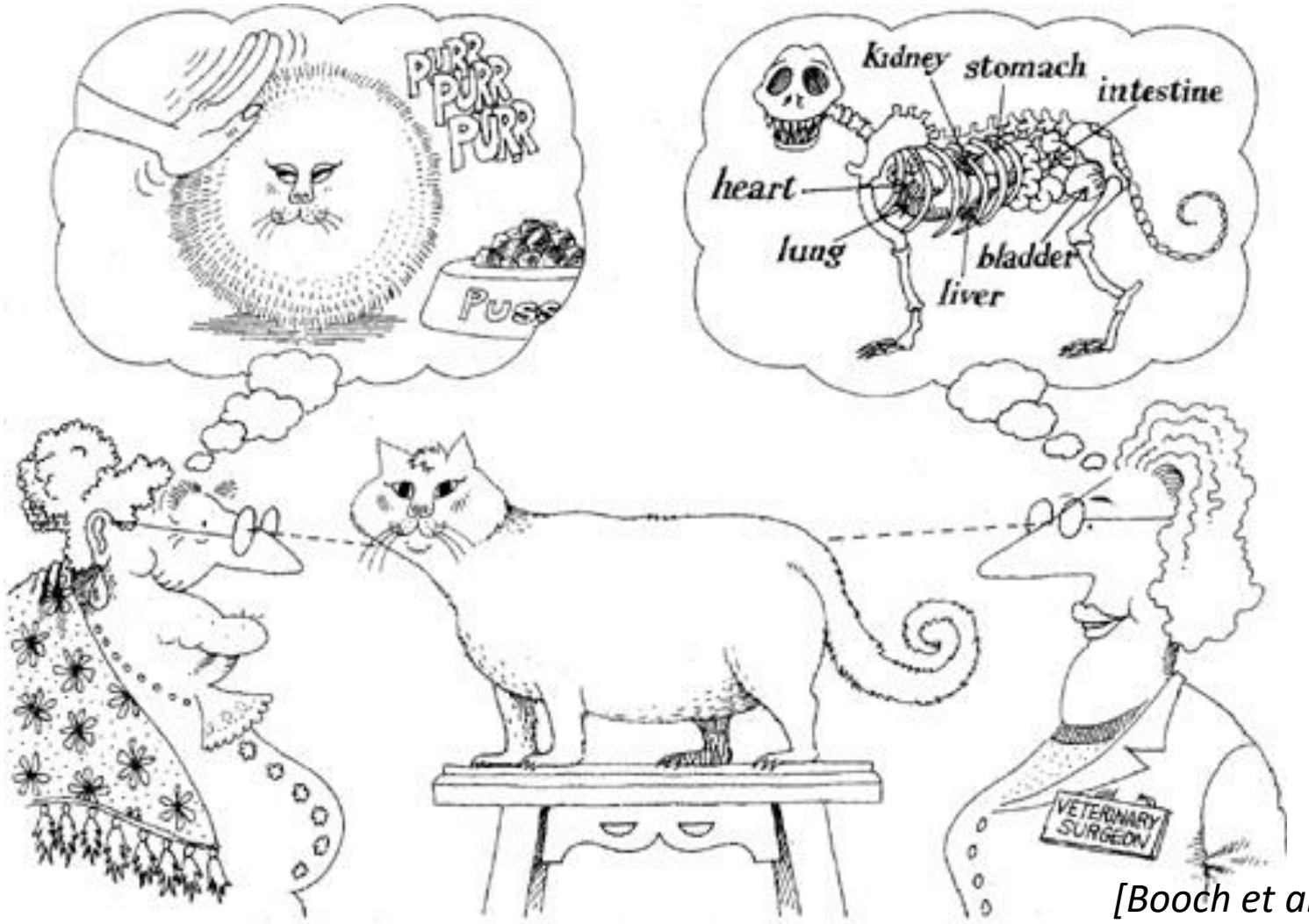
The Object Oriented Model

- The object model is the conceptual framework for all things object-oriented
- Major elements (essential for OO)
  - Abstraction
  - Encapsulation
  - Modularity
  - 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"
[Booch et. al. p.44]

Abstraction



[Booch et al., p.45]

Abstraction

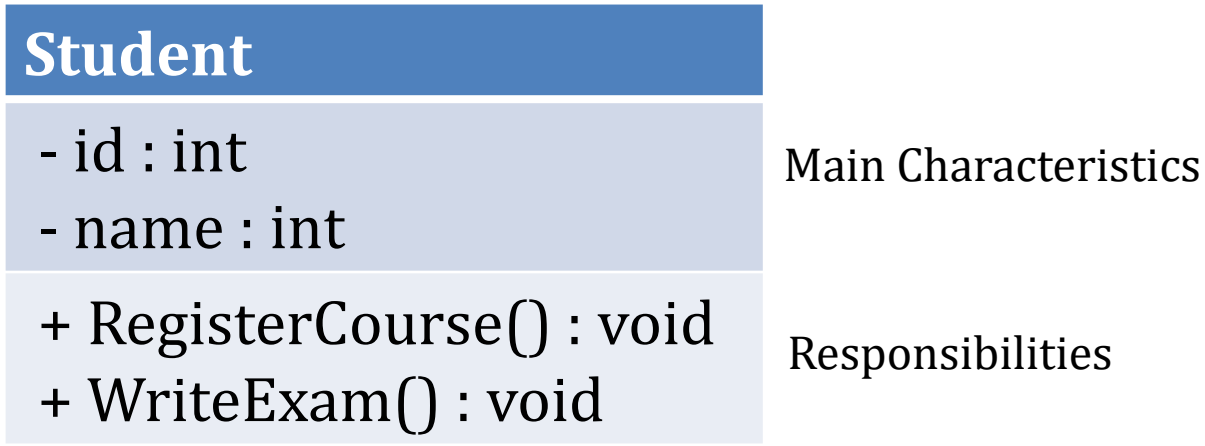
Abstraction can be achieved using:

- Principle of Least Commitment
  - Captures object's essential behavior – nothing more
- Principle of Least Astonishment
  - Capture entire object's behavior – no surprises

Why Choose Abstraction

- Object Oriented Decomposition vs Algorithmic Decomposition
  - Freedom to ignore certain details with focus on big picture design
  - Easy shift from one level of details to another
    - High Level - less details
    - Low Level - more details
  - Master complexity
- Level depends on the scope and view

Example


Student
- id : int
- name : int
+ RegisterCourse() : void
+ WriteExam() : void
Main Characteristics
Responsibilities

Recap – Abstraction

- Abstraction precedes any decisions
  - "helps people to think about what they are doing" *[Booch;2007]*
  - Allows users to extract and model essential objects in a particular application domain
  - Focuses on important properties of objects while ignoring inessential details
  - Serves as a contract between application users and application developers

Recap – Abstraction

- The level (degree) of abstraction depends on the scope chosen by the user
  - Higher level: focus on less and more important information, and fewer objects
  - Lower level: reveal more details, more information, and more objects
- Abstraction does NOT specify how information is handled
  - Abstraction does not worry about implementation

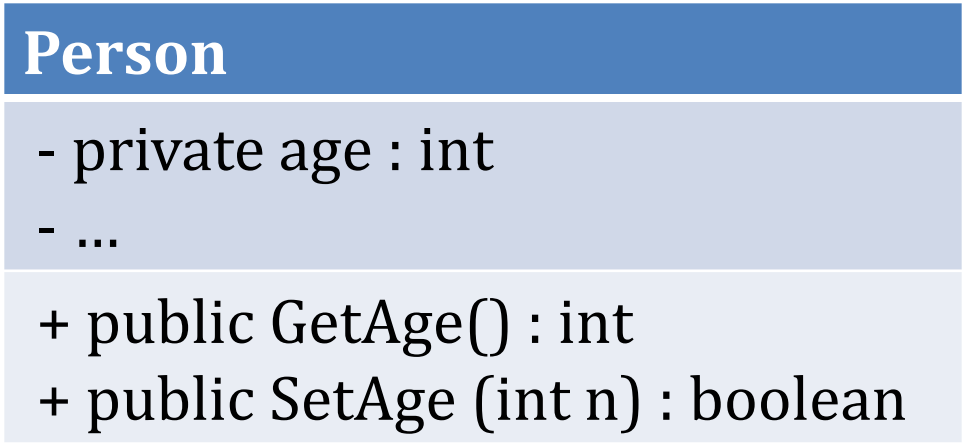
Example

```
class Person{
public:
      int age;
      … ;
};
int main(){
      Person a;
      a.age = 25;
      Person b;
      b.age = -56;
}
```

Example – Solution

- How do we solve this problem?
  - Separate Abstraction from Implementation
  - The role of Encapsulation which creates a barrier between:
    - Abstraction – handling only the public side of an object
    - Implementation – handling the private side of an object
  - Abstraction and Encapsulation are complementary
    - Abstraction focuses on the observable behavior of an object
    - Encapsulation focuses on the implementation that gives rise to this behavior

Example – Solution



- Goal 1: protect data from being randomly manipulated
  - Hide age from outside objects using access modifiers (private)
  - Restrict data access to the object's member functions
- Goal 2: control access to data by other objects
  - Bind data manipulation to the object
  - Use Getter and Setter methods to manipulate the age

Exercise

- Identify a key abstraction for the following problem:

  On a hydroponics farm, plants are grown in a nutrient solution, without sand, gravel, or other soils. Maintaining the proper greenhouse environment is a delicate job, and depends upon the kind of plant being grown and its age. One must control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations. On a large farm, it is not unusual to have an automated system that constantly monitors and adjusts these elements. Simply stated, the purpose of an automated gardener is to efficiently carry out, with minimal human intervention, growing plans for the healthy production of multiple crops.

Exercise – Solution

Abstraction: Temperature Sensor

Important Characteristics:
- Temperature
- Location

Responsibilities:
- Report current temperature
- Calibrate

Abstraction: Growing Plan

Important Characteristics:
- Name

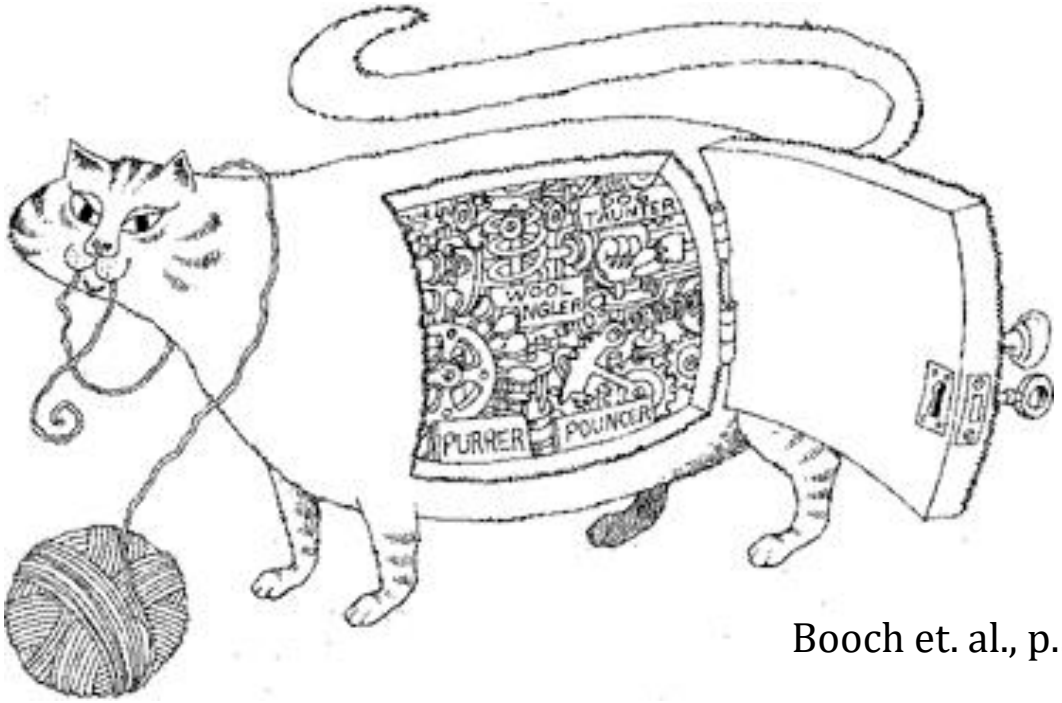
Responsibilities:
- Establish plan
- Modify plan
- Clear plan

Abstraction: Active Temperature Sensor

Important Characteristics:
- Temperature
- Location
- Setpoint

Responsibilities:
- Report current temperature
- Calibrate
- Establish setpoint

Encapsulation



Booch et. al., p.51

"Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation" *[Booch;2007]*

Encapsulation

- Encapsulation is the process of
  - Protecting the data elements from being exposed to the outside of the Object directly
  - Enforcing that an Object's data, and the functionality manipulating data must both be included within the Object
  - Relying on specialized methods to manipulate data elements within an Object
  - Leading to clear separation of concerns

How Encapsulation is Done

- Achieved through information hiding
  - Hiding the structure
    - Defining object parts independently from other parts
    - e.g., the physical representation of a collection of data items is left as an implementation detail (list, tree, set, …)
  - Hiding the implementation of the methods
    - Defining interfaces to objects without committing to a particular implementation
    - e.g., processing data in a certain order (Merge sort, Heap sort, Quick-sort) to achieve a certain behavior (sort)

How Encapsulation is Done

In practice each Class has two parts:

- Interface
  - The outside view of the Class as well as its behavior
  - The protocol for accessing the class's internal data and its operations
- Implementation
  - Representation of the Class abstraction as well as the operations that achieve the expected behavior
  - Encapsulates internal details with no further assumptions

Example – Solution

```
#include<iostream>
class Person {
private: int age;
public:
        int GetAge() {
                return age;
        }

        bool SetAge(int n) {
                if (n > 0 && n < 200) {
                        age = n;
                        return true;
                }
                return false;
        }
};
int main() {
        Person a;
        a.SetAge(25);
        std::cout<<a.GetAge();
        return 0;
}
```

Exercise

- Write code for a program that prints out the area of a square given its dimension
  - Use abstraction and encapsulation techniques
    - Identify the main characteristics and responsibilities
- Can you change the program so that the area calculated is of a circle with the given dimension instead of a square
  - Where would you make changes?
  - Would you need to make changes in main()?
  - Was the abstraction and encapsulation implemented correctly?

Solution 1

```
#include<iostream>
class Square {
private: float side;
public:
    void SetSide(float x) {
        side = x;
    }

    float GetArea() {
        return side*side;
    }
};

class Circle {
private: float radius;
public:
    void SetRadius(float x) {
        radius = x;
    }

    float GetArea() {
        return 3.14159*radius*radius;
    }
};
```

```
int main() {
    Circle c;
    Square s;
    int i;
    std::cin>>i;

    c.SetRadius(i);
    s.SetSide(i);
    std::cout<<"Area of circle = "<<c.GetArea()
            <<"\nArea of square = "<<s.GetArea();
}
```

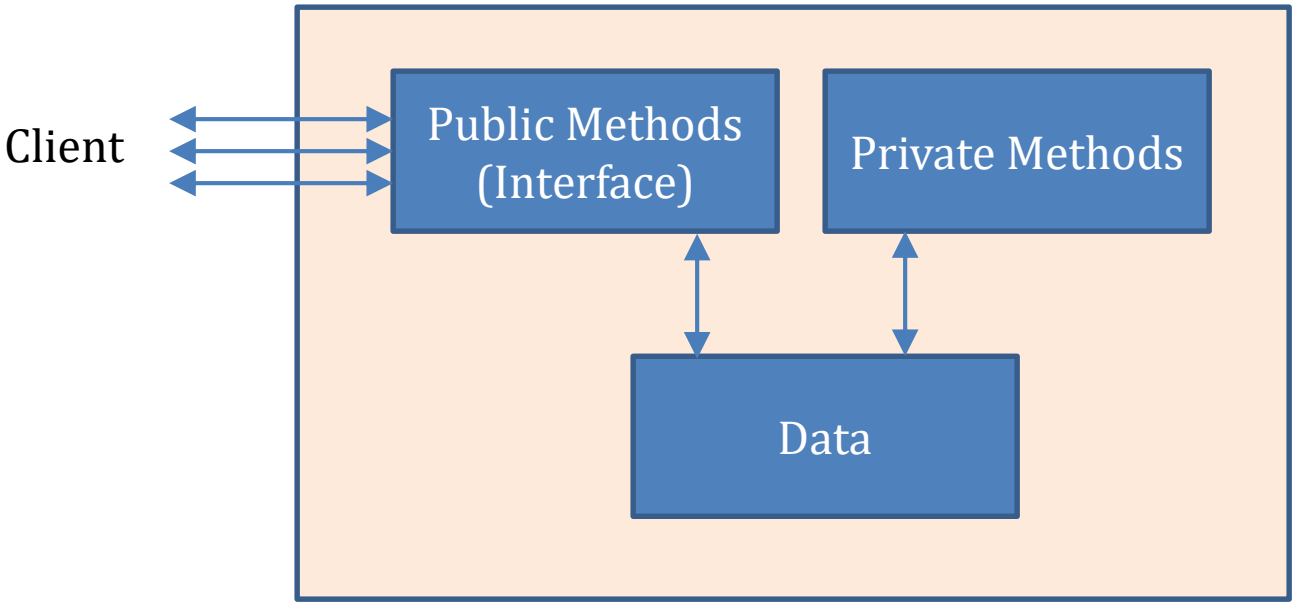
Solution 2

```
#include<iostream>
class Shape {
private: float dim;
public:
    void SetDim(float x) {
        dim = x;
    }

    float GetAreaSq() {
        return dim*dim;
    }

    float GetAreaCirc(){
        return 3.14159*dim*dim;
    }
};
```

```
int main() {
    Shape sh;
    int i;
    std::cin>>i;

    sh.SetDim(i);
    std::cout<<"Area of circle = "<<sh.GetAreaCirc()
            <<"\nArea of square = "<<sh.GetAreaSq();
}
```

Why Encapsulation?

- Flexibility, maintenance and extensibility
  - Allow changes in implementation without disturbing users of an Object



- What about reusability?

Why Encapsulation

- Data protection
  - Eliminate the need to check that object's data is being manipulated by proper functions
  - Encapsulation binds the functions manipulating an object's data to the object
- Access Control
  - Define which parts of an object (data, functions) can be accessed by other objects

Abstraction, Encapsulation – Example

For the hydroponics farm discussed before, an object participating in the automatic gardening system was a nutrient dispenser, used by a growing plan to control the relative amounts of different chemicals mixed together in a particular plant's nutrient solution. Given the following description, what might the abstract interface consist of? What should be encapsulated within the object?

A nutrient dispenser is created for every vat of nutrient solution, and initialized with the locations of the chemical reservoirs over which it has control. Periodically, the growing plan checks the state of the nutrient solutions for the various plants. If a solution has moved beyond the acceptable range, the appropriate nutrient dispensor is asked to correct the problem, receiving information concerning the current solution profile and the target concentrations. The dispensor analyzes the difference between the two readings and injects the chemicals needed to reach the desired level. The dispensor can also report the current reservoir level of any chemical it controls.

Example – Sample Solution

- Software dispenser object:
  - Abstract interface
    - Create (chemical_name, location for adjusting vat's solution)
    - Destroy
    - CurrentLevel (chemical_name)
    - Adjust (curr_sol_profile, des_col_profile)
  - Encapsulation:
    - Association between chemical name and physical valve
    - Algorithm for chemical composition adjustment
    - Mechanism by which it injects chemicals
    - How the dispenser is turned on/off
    - Means for detecting current chemical levels

Encapsulation

- What to do when the abstraction model becomes very large?
  - Breaking down complex problems into more manageable sub-problems is always a good strategy
  - ... the role of Modularity

Modularity

- Object Oriented approach of breaking down complex problems into more manageable sub-problems
- Allows grouping a set of classes into discrete units: Modules
  - Allow well defined boundaries within an application's Object Model
  - Serve as physical structure of an application
  - Modules are packages in Java, namespaces in C++
- Modules are essential to managing complexity of applications with hundreds of classes
  - Makes software easier to understand and maintain
  - Helps reduce software cost

Modularity

- "Modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules" *[Booch et. al. p.55]*

Booch et. al., p.54

What is a Module?

- Physical container for classes and objects
  - Logically related to perform a certain functionality within an application
  - e.g., package java.io groups together all classes that perform input and output operations including reading and writing to a file system, data streams, etc.
- Choosing the right set of modules is not an easy task
  - Detect logically related classes and objects
  - Select which elements can be exposed to other modules
  - ... as difficult as choosing the right set of abstractions

Modularization Design

"Arbitrary modularization is sometimes worse than no modularization at all"*[Booch;2007;p:55]*

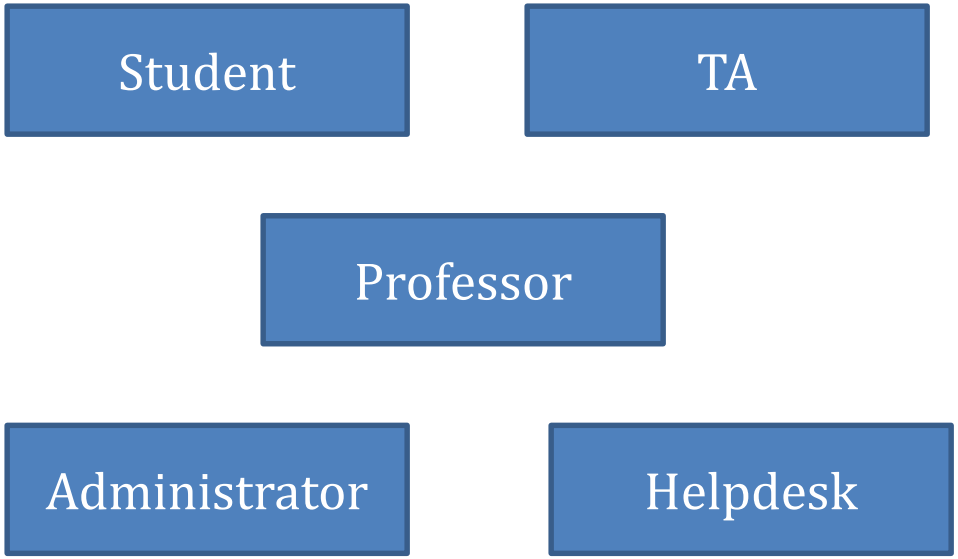
- Poor design
  - Group all classes and objects in the same module
- Good design
  - Group logically related classes into the same module
  - Selectively choose what to expose to other modules

Modularization Guidelines

- Simplicity: Choose modules that are simple enough to understand without knowledge of other modules
- Independence: Implement modules such that code maintenance does not affect other modules
- Narrow Interface: Choose small interfaces for modules without affecting other module's need
- Balance of concerns
  - Encapsulation: Hide system details that are likely to change independently
  - Abstraction: Allow assumptions by exposing details that are unlikely to change
- Reuse: Group classes in a way to facilitate their re-use

Modularization Example

- DEPARTMENT module



Why Modularize?

- In general modules minimize software cost in case of revisions
- Cohesive modules group objects that are logically related
- Loosely coupled modules minimize dependencies among modules

Modularization Example 2

A new library at Dal is developing a new browsing and request system for its holdings. Based on the project description below, what high-level decomposition of the work into modules would you propose and why?

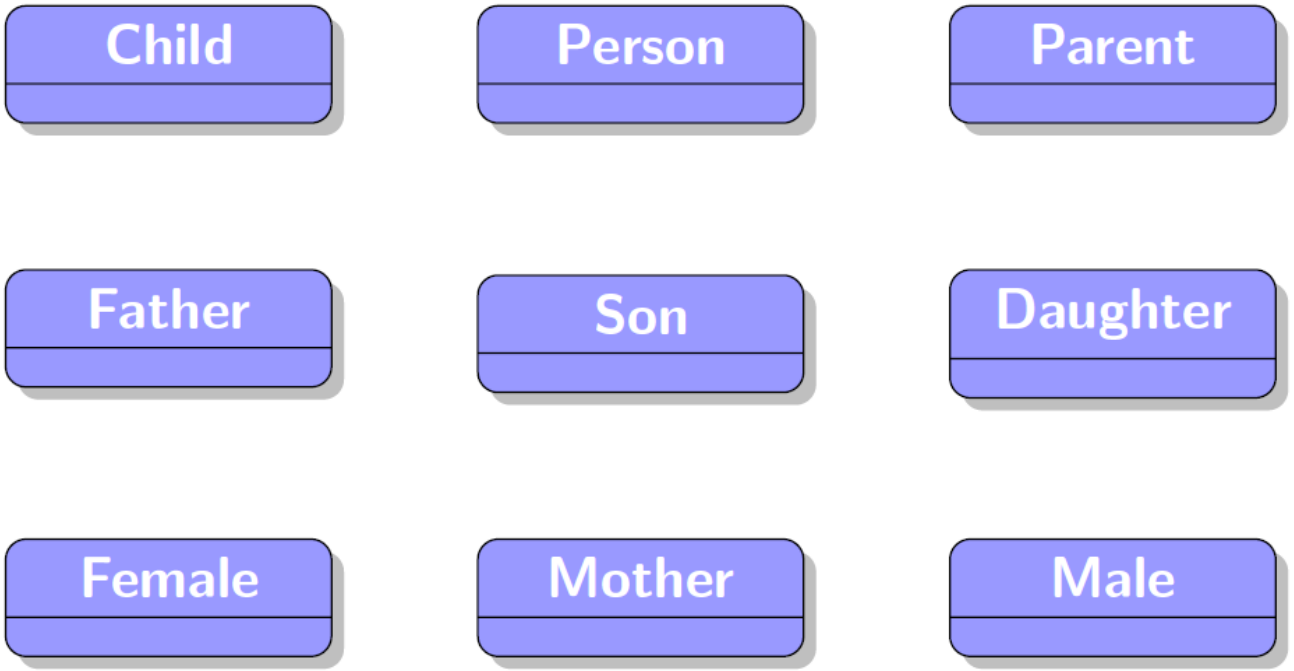
The user interface to the system will run on two platforms, the 10 Mac workstations available in the library itself, and the PC-clone machines that the undergraduates use as home terminals; all of the database programs will run on the university mainframe. The students will be provided with a simple interface that allows them to browse the library database much like a card catalog; as they browse, they can request any item, and if it is available it will be forwarded to them. The PC interface will be menu-based; the menu design (wording, sequencing, etc.) is being developed as a class project in an undergraduate course on human-computer interaction. The Mac interface will be a direct manipulation system used by students working in the library. Librarians will be using an extended version of this Mac system to maintain the library database, to track usage, and to send out overdue notices. The project will build on an already existing database of library holdings, but will enrich the current classification system to support more flexible browsing.

Modularization Example 2 – Sample Solution

- PC interface module
  - Constituent menus
    - Developed and iterated independently in the class project
  - Database
    - New shell providing enhanced indexing capability
    - Contain the original database as a submodule for clean separation
- Application programs
  - Student tasks (browsing and requesting)
  - Librarian tasks (library management)
- Network traffic module
  - To handle network traffic between user workstations and the database programs

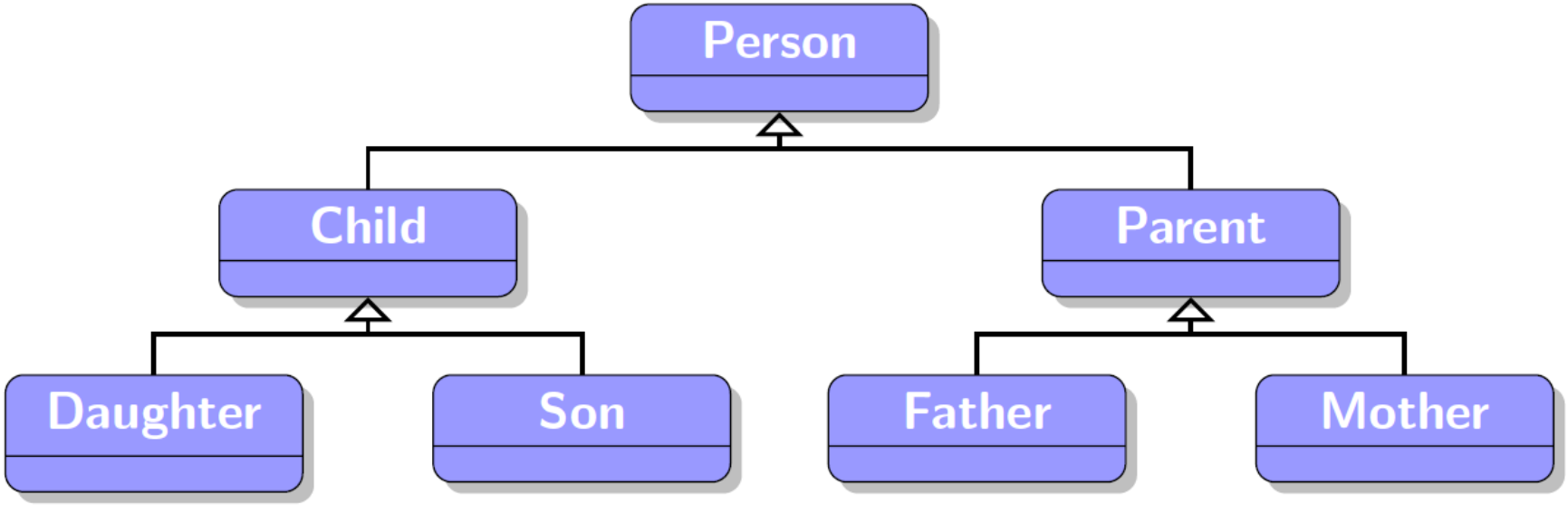
Motivation

- What else can we find out about classes to facilitate understanding?



Motivation

- Observation: most complex systems are organized by associating a partial order on the objects and abstractions within it
- Abstractions can be organized into hierarchies



Hierarchy

- Abstractions also need to be organized (not only grouped together)
  - Complex systems are usually organized based on an orderly relation between their abstraction model
  - How can such ordering be achieved in an Object-Oriented approach?
    - Using hierarchies

What is a Hierarchy?

- A hierarchy is a ranking or ordering of abstractions
- Detecting hierarchies between objects can facilitate the understanding of the problem

Booch et. al., p.59

Types of Hierarchies

- Two main types of hierarchies
  - "**is a**" Hierarchy – Inheritance relation between classes, also known as "**kind of**"
  - "**part of**" Hierarchy – Aggregation relation between objects

Example

Characterize the different hierarchical relationships implied by the following descriptions:

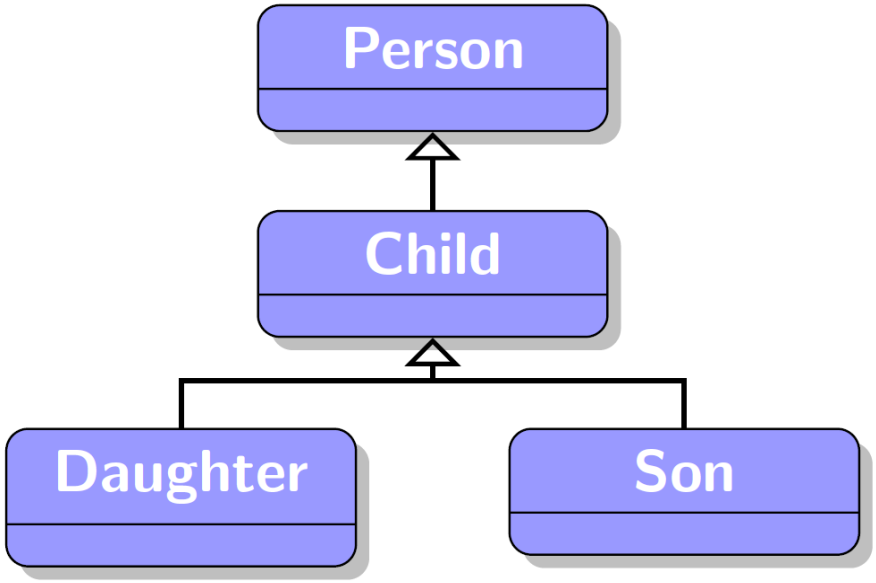
1) A baseball team consists of a manager and 25 players. Each player normally plays a single defensive position (e.g., pitcher, catcher), but some players are utility players who can serve in a range of positions.
   - Manager and 25 players “part of” team
   - Defensive position “part of” player
   - Utility defensive position “is a (kind of)” two or more single positions

Example

2. The main window of a library book browser has a title bar and three subpanes, one for selecting book categories, another for choosing particular books within a category, and a third for displaying an abstract of the book. Each subpane has its own scrollbar.
   - Menu bar and subpanes "part of" window
   - Scrollbar "part of" subpane
   - Category, title selection, and abstract display "is a (kind of)" subpane
3. A heating control system has three separate thermostats. Two of the thermostats are basic thermometers that must be adjusted manually, but the third can be adjusted either manually or by setting a timer.
   - Thermostats are "part of" the control system
   - Thermostat "is a" thermometer
   - Timer control is "part of" the specialized thermometer

Inheritance – Class Structure

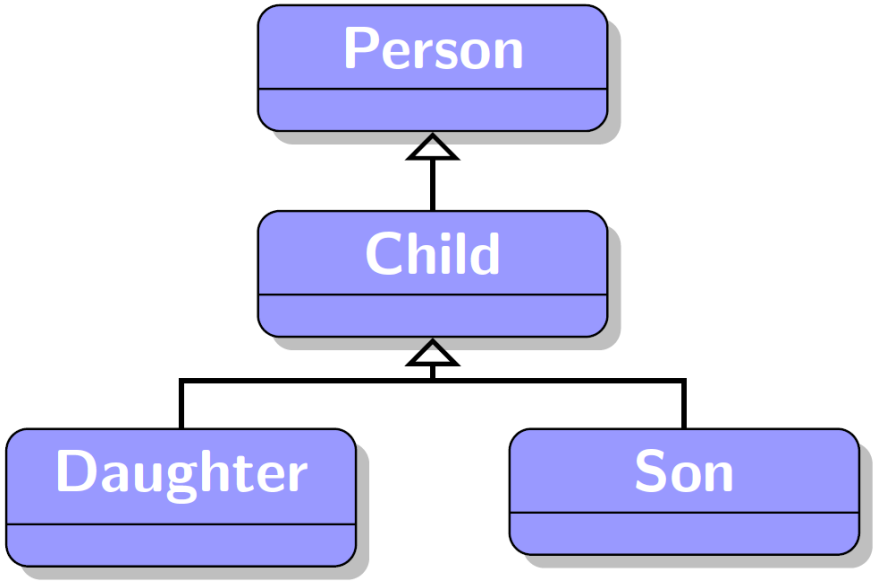
- Generalization – Superclass
  - person is a generalization of child
- Specialization – Subclass
  - daughter is a specialization of child



Single Inheritance

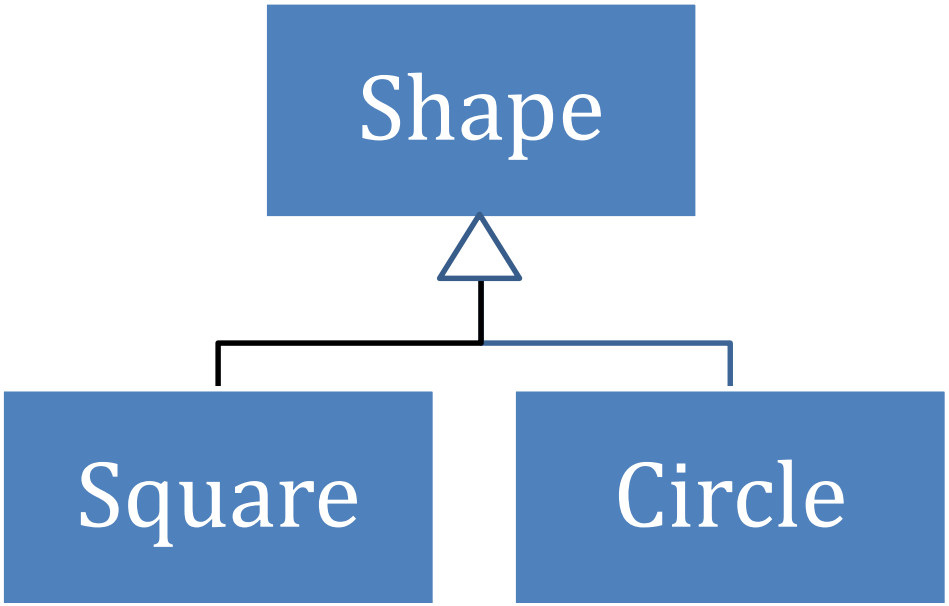
With single inheritance relations, a subclass A

- Has exactly ONE superclass B
- Inherits all properties of B
- Provide additional properties
- B may also be a subclass of some class C



Solution 3

- Calculating area of shapes



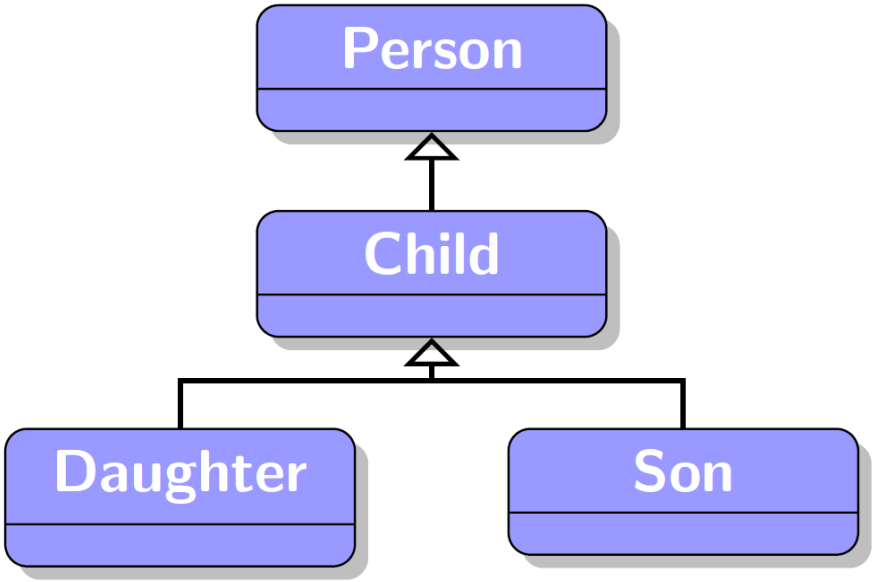
Solution 3

```
#include<iostream>

class Shape {
protected: float dim;
public:
    void SetDim(float x) {
        dim = x;
    }
};

class sq : public Shape {
public:
    float GetArea() {
        return dim*dim;
    }
};

class circ : public Shape {
public:
    float GetArea(){
        return 3.14159*dim*dim;
    }
};
```

```
int main() {
    float i;
    std::cin>>i;

    circ c;
    sq s;
    c.SetDim(i);
    s.SetDim(i);
    std::cout<<"Area of circle = "<<c.GetArea()
            <<"\nArea of square = "<<s.GetArea();
}
```

Exercise

- Write down the code to implement the structure shown below



Exercise – Sample Solution

```
class Person
{
private:
    std::string name;
public:
    std::string GetName()
    {return name;}

    void SetName(std::string nm)
    {name = nm;}
};

class Child : public Person
{};

class Daughter : public Child
{};

class Son : public Child
{};
```

```
#include<iostream>
#include<string>

int main()
{
    Person p;
    p.SetName("Khurram");
    std::cout <<p.GetName();
}
```

What does a Derived Class Inherit?

- It inherits all the accessible members of the base class
- Accessibility also depends on the access specifier
  - e.g. class child_class : **public** base_class
    - Child_class inherits all members of base_class with the same level of accessibility that they had in the base_class
  - Could also be **protected**
    - All public members of base_class are inherited as protected in the child_class
  - Could also be **private**
    - All public and protected members of base_class become private in the child_class

What does a Derived Class NOT Inherit?

- Derived class does not inherit:
  - Constructor and destructor of the base class
  - Private members of the base class
  - Friends of the base class
  - Its assignment operator members (operator=)

Note that the constructor and destructor of the base class are not inherited, however, they are automatically called by the constructor and destructor of the derived class

Example

```
class Person
{
private:
    std::string name;
public:
    std::string GetName()
    {return name;}

    void SetName(std::string nm)
    {name = nm;}
};

class Child : public Person
{};

class Daughter : public Child
{};

class Son : public Child
{};
```

```
#include<iostream>
#include<string>

int main()
{
    Person p;
    Daughter d;
    d.SetName("Aniya");
    std::cout <<d.GetName();
}
```

What is the output?

Aniya

Example

What is the output?

```
int main()
{
    Person p;
    Daughter d;
    d.SetName("Aniya");
    p = d;
    std::cout <<p.GetName();
}
```

Aniya

```
int main()
{
    Person p;
    Daughter d;
    p.SetName("Khurram");
    d = p;
    std::cout <<d.GetName();
}
```

Error!

```
int main()
{
    Son s;
    Daughter d;
    d.SetName("Aniya");
    s = d;
    std::cout <<s.GetName();
}
```

Error!

Example

```
class Person
{
protected:
    std::string name;
    void SetName(std::string nm)
    {name = nm;}
public:
    std::string GetName()
    {return name;}

};

class Child : public Person
{
    public:
    void SetChildName()
    {
        SetName("Aniya");
    }
};
```

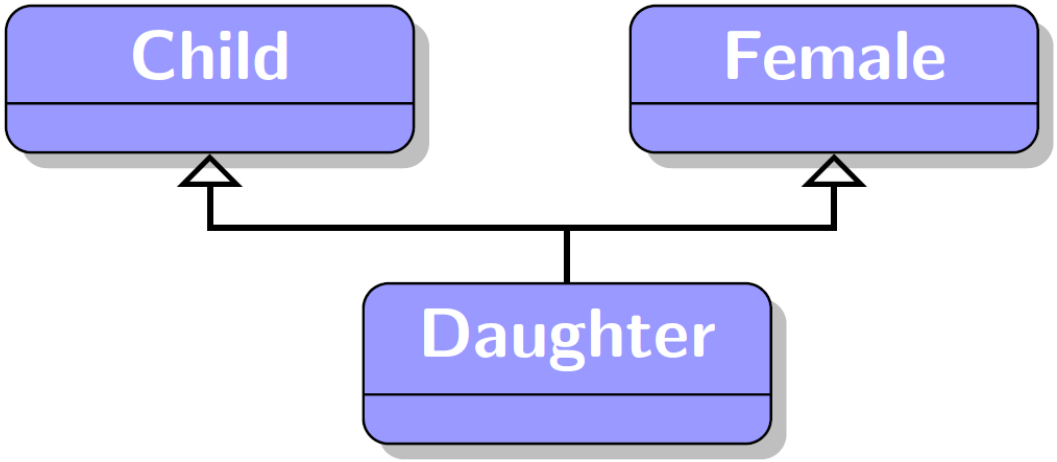
```
#include<iostream>
#include<string>

int main()
{
    Child c;
    c.SetChildName();
    std::cout <<c.GetName();
}
```

Aniya

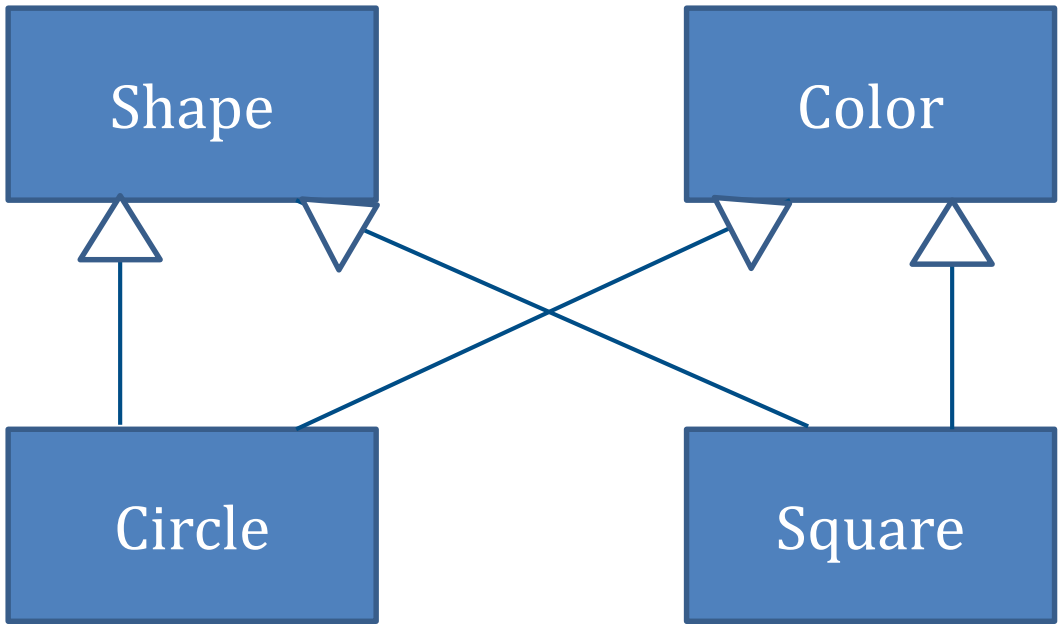
Multiple Inheritance

With multiple inheritance relations, a subclass A:

- Has more then one superclass
- Inherits all properties of all its superclasses
- Avoids redundant declarations of properties



Example



The “shape” Class

```
#include<iostream>
#include<string>

using namespace std;

class shape {
protected:
   float dim;
public:
   shape() {
       dim = 1;
       cout << "\nDefault dimension is "<<dim<<endl;
   }
   void SetDim(float n) {
      dim = n;
   }
};
```

The “color” Class

```
class color {
protected:
	string m_clr;
public:
	void SetColor(string l_clr) {
		m_clr = l_clr;
	}
	color() {
		m_clr = "White";
		cout << "\nDefault color is WHITE\n";
	}
	string GetColor() {
		return m_clr;
	}
};
```

The “sq” class

```
class sq : public shape, public color {
      public:
      float CalcArea() {
            return dim*dim;
      }
      sq() {
            m_clr = "Green";
            cout<< "\nDefault square is GREEN\n";
      }
};
```

The “circ” class

```
class circ : public shape, public color {
      public:
      float CalcArea() {
            return 3.14159*dim*dim;
      }
      circ() {
            m_clr = "Red";
            cout << "\nDefault circle is RED\n";
      }
};
```

The “main” function

```
void main() {
	float n;
	string clr;
	cout << "Enter dimension: ";
	cin >> n;
	cout << "Enter color: ";
	cin >> clr;
	circ c;
	sq s;
	c.SetColor(clr);
	s.SetDim(n);
	cout << "\nArea of the "<<c.GetColor()<<" Circle is: "
<< c.CalcArea() <<"\n";
	cout << "\nArea of the "<<s.GetColor()<<" Square is: "
<< s.CalcArea() << "\n";
}
```

Output

```
Enter dimension: 2
Enter color: BLUE

Default dimension is 1

Default color is White

Default circle is Red

Default dimension is 1

Default color is White

Default square is Green

Area of the BLUE Circle is: 3.14159

Area of the Green Square is: 4
```

Multiple Inheritance

Beware of!

- Name clash: more than one superclass has the same name for a field or operation as a peer superclass
- Repeated inheritance: more than one superclass shares a common superclass
- Overuse: wrong inheritance relation. e.g., cotton candy is not a kind of cotton!

Multiple Inheritance – Example

- As part of a military video game, a designer has created a vehicle class hierarchy.
- The base class is Vehicle, and it has AirVehicle, LandVehicle and SeaVehicle as derived classes.
- The class SeaPlane inherits from both AirVehicle and SeaVehicle.
- Specifically, what design issues had to be considered in developing the SeaPlane class?

Multiple Inheritance – Example Solution

- Multiple inheritance with a repeated inheritance issue:
  - Two superclasses part of the same hierarchy, hence, one or more baseclasses will be inherited redundantly
  - Possible name clashes
    - Move() -> as an air vehicle or sea vehicle?

Inheritance and Encapsulation

- A subclass can
  - access data related to its superclass
  - call a protected function of its superclass
  - refer directly to a superclass of its superclass
    - → encapsulation violated!
- Encapsulation can be enforced
  - **private** - data that can be accessed by the class itself
  - **protected** - data that can be accessed by the class and its subclasses
  - **public** - data that can be accessed by any other class

Why use Inheritance

- Economy of expression: reduce code duplication and complexity
- Organization of abstractions: detect logical relations between classes
- Consistency: organize and extend abstractions in a consistent way