Object Oriented Methodologies

Dr.RanjanaVyas

Indian Institute of Information Technology, Allahabad

OOM

- Think in Object Oriented manner !
    - *Objects rather than Procedures…*
- Model /Design in Object Oriented Methodology !
    - *Use UML diagrams for Model-Design…*
- Develop in Object Oriented Programming !!
    - *Use of Java programming…..*
- Maintain/ReUse using Object Oriented Features !!
    - Use of JAR, Design Patterns etc…

OOM for Software Projects

- **Software Projects & Design: Design Patterns**
  - ***OOM Design, UML & GoF dictum***
  - ***Analysis Model to Design Model***
  - ***Patterns in Engineering: Concepts wrt Software Engineering***
    - ***Why we need good Design..& What is the good Design..?***
    - ***What are Design objectives..?***
    - ***UML-Unified Modeling Language***
  - ***Design Patterns: GoF and after effects….***
- **Design Principles: Design & OO relationships**
  - ***Inheritance, Composition & Aggregation***
    - **Advantages and Disadvantages of Inheritance**
    - **How to Create Quality Classes**
    - **Design Patterns: Concepts**
- **OOM: GUI, Applets & Mobile Apps**
  - **Implementation issues**
- **Recent trends in OOSE**

OOM

It’s believed that Java is more effective in Large & Mid sized software projects and Web based applications, specially involving Large number of classes and thus design..

- ➢ If we develop big Software project with OOM, then what’s the significance of design ?

How will be OOM useful in Design..?

- ν UML- Unified Modeling Language…..!!
- ν OOM & Software design..
- ν What exactly GoF contributed in OO Design !!

OOM for Software Projects

- As OOM has become de-facto software paradigm today, it is required for medium sized or large sized project follow **Software Engineering approach** for successful software project implementation.
  - One can not directly jump right into the coding phase, without spending adequate time on Understanding the Requirement Analysis of Software, Modeling & Design of the project.
  - And then only Implementation of the agreed Design.
- A group of approaches are developed for Object Oriented Software development….
- **OO provides many specific design** principles based on Object Oriented characteristics, these principles are implemented using **UML**-Unified Modeling Languages

OO Design

- **Software Design** in general deals with having specific design objectives being fulfilled to have Software with
  - Flexibility
  - Maintainability
  - Usability
  - Highest performance
- **General/Structured software design** emphasizes on design of software modules with functional independence
  - Cohesion
  - Coupling
- **OO provides many specific design** principles based on Object Oriented characteristics

Modeling, Design & GoF

Java GUI, Applets & Apps development

Modeling & Design in Engineering

If a Car company has been asked to design Racing car Wheels then what should Company do…start with Laws of Physics and Automobile Engineering or study the existing design specifications of existing Wheels and incorporate desired changes in the Wheels..

Software Design & Standard Design Patterns

- Which Class is to become Parent Class, Which are to be chosen as Child Class..?
- Do we always need to design Software with designing various software modules such that we achieve design objectives or can we have some standard design patterns ??
- Pattern-based design was introduced into architecture and engineering in the 1950's
- Almost immediately, software engineers began using patterns for designing software
- It wasn't until a Group of Four professionals combined forces that pattern-based design became well-known and commonplace
  - This group was known as the Gang of Four (GoF)

GoF has significant impact on OO practices and Researches

The Concept of **OO Design Patterns** was introduced by **GoF** and became instantly popular worldwide. The book has the first two chapters exploring the **capabilities and pitfalls** of object-oriented programming, and the remaining chapters describing 23 classic software design patterns.

GoF & OO Software

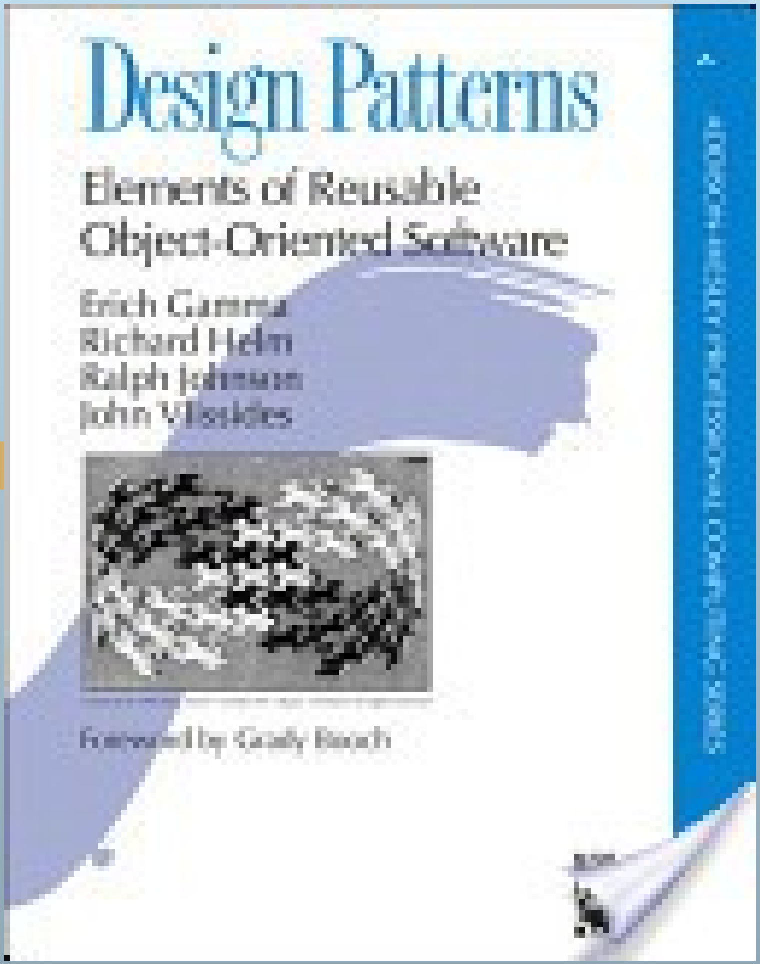
- **GoF** proposed many useful techniques related to Object Oriented Software in the popular Book
  - Authors **Erich Gamma, Richard Helm, Ralph Johnson** and **John Vlissides** with a foreword by Grady Booch. They are often referred to as the ***Gang of Four***, or *GoF.*

- The book has been **influential to the field of OO Software** and is regarded as an important source for object-oriented design theory and practice. More than **500,000** copies have been sold in English and in **13 other languages**.
  - Effective **software design** requires considering **issues that may not become visible** until later in the implementation and thus **Principles of OO Design** and Design Patterns provides a **tested Solution paradigm** to many often occurring situations.
- What are the **significant influences of GoF** in Object Orientation?

GoF-Design Patterns

- **Design Patterns: Elements of Reusable Object-Oriented Software**
  Book by Ralph Johnson, John Vlissides, Richard Helm, and Erich Gamma

- Published: October 21, 1994
  - Describing 23 classic software design patterns.
  - Regarded as an important source for **object-oriented design theory and practice.**



Shall we expect next GoF from IIIT..??

Looking at enormous talent in our students…
We can definitely keep the hope..

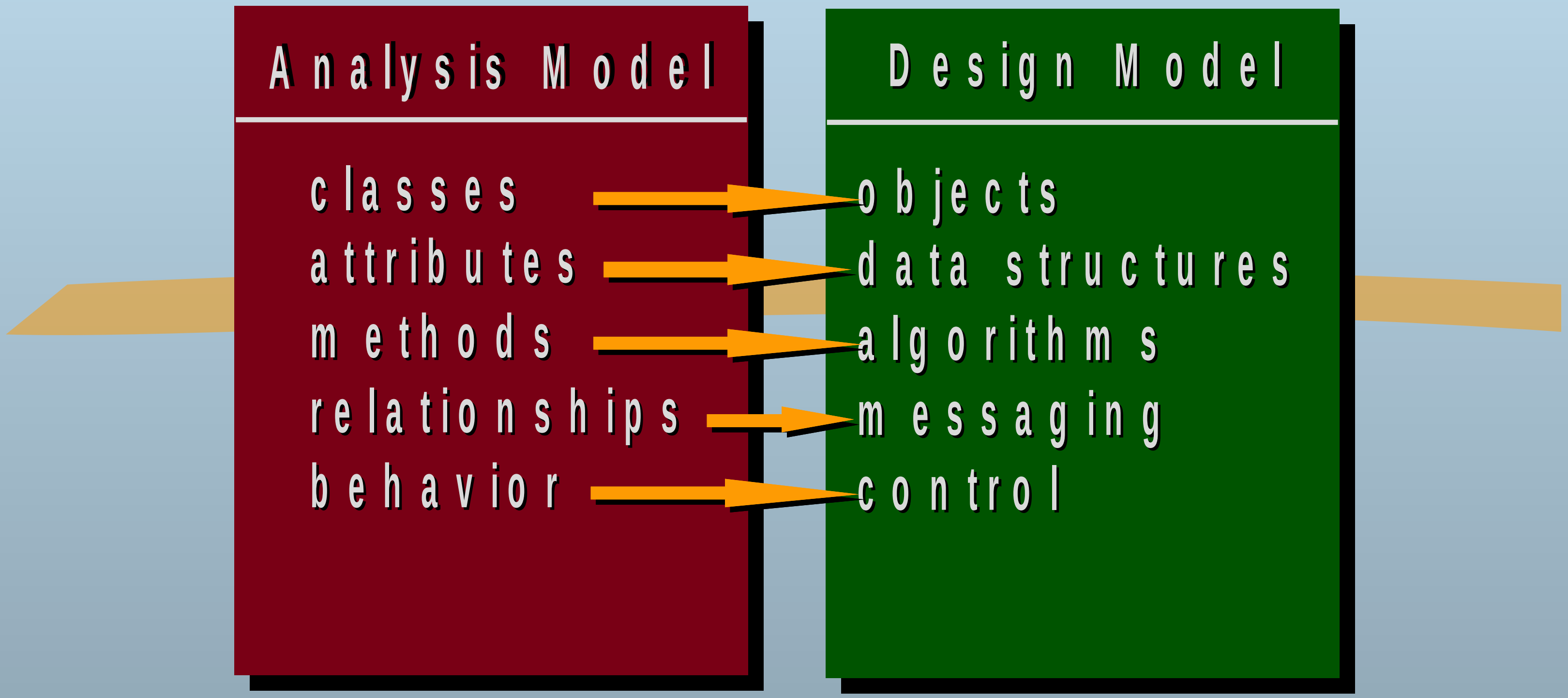
GoF Dictum…

- GoF based on their working experience proposed some significant concepts;
  - *"Favor '<u>object composition</u>' over '<u>class inheritance</u>'." (Gang of Four 1995:20)*
  - *"Program to an 'interface', not an 'implementation'." (Gang of Four 1995:18)*
- The GoF refer to <u>inheritance</u> as ***<u>white-box reuse</u>***, with white-box referring to visibility, because the internals of parent classes are often visible to <u>subclasses</u>.
- In contrast, the authors refer to <u>object composition</u> as ***<u>black-box reuse</u>*** because no internal details of composed objects need be visible in the code using them.

OO Design

- Knowing concepts like **Abstraction, Inheritance, Polymorphism** is necessary but not sufficient to make you good OO Designer !
  - A designer has objectives of creating Flexible designs that are maintainable with appropriate Re-Use provisions.
  - How can we **make Software fulfilling** above attributes?
  - How to proceed from the **OO Analysis to Design**

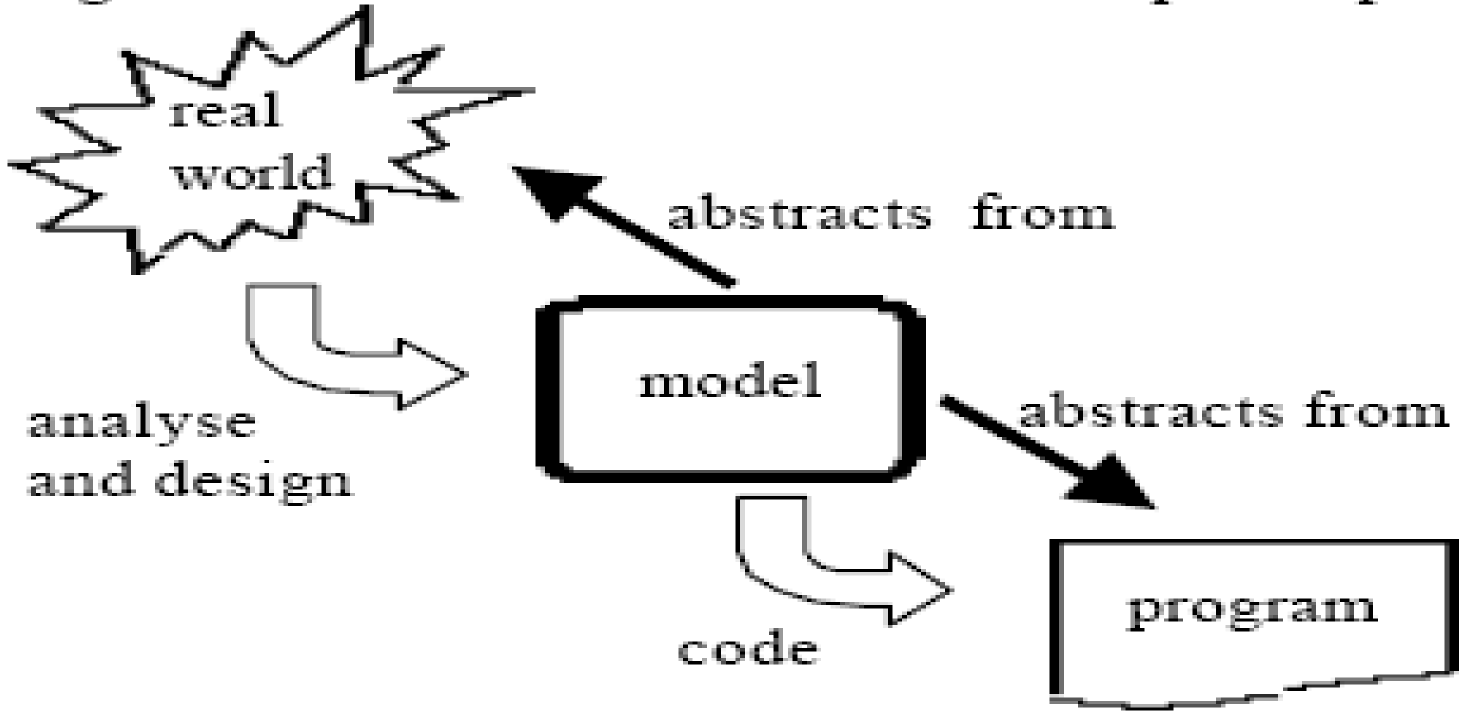
OOA to OOD



Analysis Model
classes
attributes
methods
relationships
behavior
Design Model
objects
data structures
algorithms
messaging
control

Real World: Modeling to Design

Figure 1: Role of model within development process



Design in OO & Class Relationships

Though there are many OOM characteristics but when asked to design the system, we tend to consider Inheritance first and think that we should design SuperClass and SubClass structure

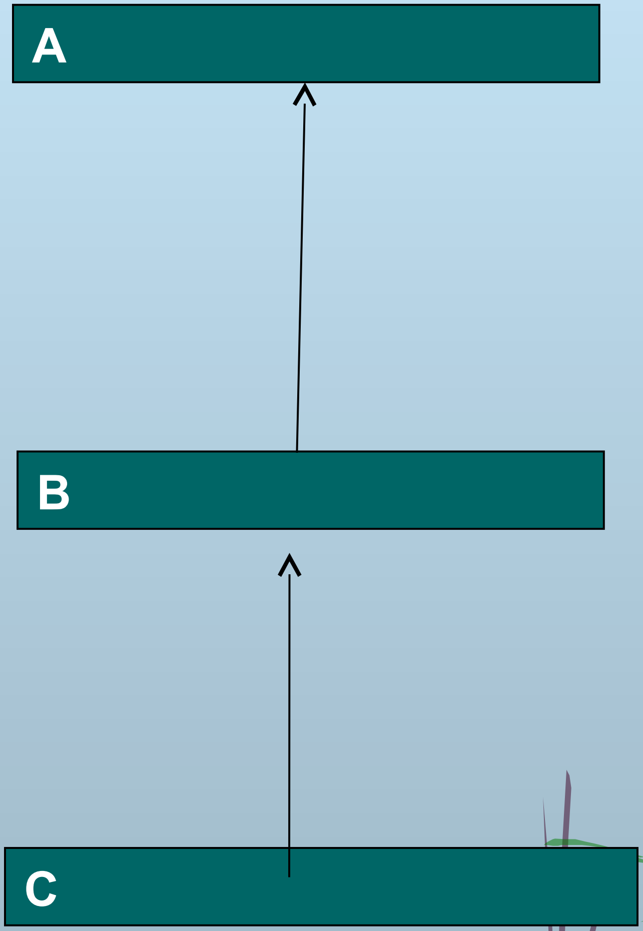
But we need to study the problem domain with other design possibilities of existing OO relationship and accordingly incorporate desired changes..

OOM Features

- Generally OO software designers model the given situation in terms of OO characteristics like Inheritance…
- Can we model the situation ONLY as Inheritance…
- Can we have some other relationships between classes other than as Inheritance ?
- What about Aggregation or Composition ???

Objects instantiation in Multi level Inheritance

- Consider a situation when we have **Multi level Inheritance** and Object instantiation;
  - **Objects of Class A with reference type A.**
  - **Objects of Class B with reference type B and also of type A.**
  - **Objects of Class C with reference type C and/or of type B , type A.**
- ***In OOP*** **the ability of an object to have many forms, which is a direct result of inheritance is known as Polymorphism.**



OOM Design

- Inheritance
  - When & Why ?
- Composition
  - What & How?
- Aggregation
  - What & How ?

Inheritance Vs Composition

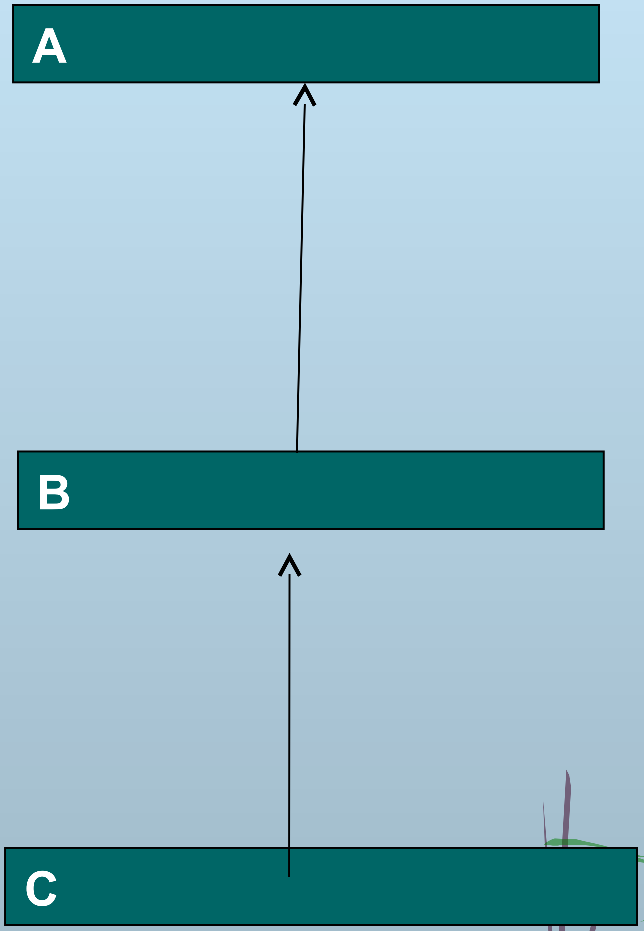
- One of the fundamental activities of an object-oriented design is establishing relationships between classes. Two fundamental ways to relate classes are
  - *inheritance* and *composition*
- Composition is also one of the effective ways of relating Classes.
- Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get **the functionality of inheritance** when you use composition.
- Why **Composition ?**
- Are there any specific *disadvantages* in Inheritance..?

Inheritance and Code Reuse..

- Using inheritance to *achieve code reuse suffers* from the following problems:
  - **You cannot change the reused behaviour at runtime. Inheritance is a compile-time dependency, You cannot change this at runtime.**
  - **You cannot replace the reused behaviour from the outside for the sake of testing..**
  - **You are dependent on multiple inheritance for all but the most simple applications**. This opens the door for diamond shaped inheritance trees which increase the code complexity a lot.
- **Preferring composition over inheritance to reuse code avoids all these issues.**

Objects in Multi level Inheritance

- Consider a situation when we have **Multi level Inheritance** and Object instantiation;
    - **Objects of** Class A **will have all the attributes and methods that may be common to all of** Class B**.**
    - **Objects of** Class B **instantiated with reference** type B **and also of** type A **will have all the attributes-Methods of** Class A
    - **Objects of** Class C **with reference** type C **or of** type B , type A will have all that's in Class A and Class B..
- ***If now wish to change something in*** **Class A** **then all the classes B &C will be effected as direct result of inheritance…and if we have similarly many more child classes then..?**



Dynamic binding, polymorphism & change

- When you establish an inheritance relationship between two classes, you get to take advantage of ***dynamic binding*** and ***polymorphism***.
  - In Polymorphism we can use a **variable of a superclass type to hold a reference to an object whose class is any of its subclasses**.
- Dynamic binding means the JVM will decide at runtime which **method implementation** to invoke based on the class of the object.
- In inheritance**, superclasses** are often said to be "**fragile**," because one little change to a superclass can ripple out and require changes in many other places in the application's code.
- It turns out that when you change in Superclass and you are not careful about its effect in Subclasses, there is possible error and this does not yields easier-to-change code.

OOM Design

- Inheritance
  - When & Why ?
- Composition
  - What & How?
- Aggregation
  - What & How ?

Use inheritance only when all of the following are satisfied:

- A subclass expresses "is a special kind of" relationship but not “has a “ relationship.
  - A subclass extends, rather than overrides or nullifies, the responsibilities of its superclass.
  - For a class in the actual Problem Domain, the subclass specializes a role, transaction or device

Inheritance vs. Composition

- Inheritance means that one class inherits the characteristics of another class.
  This is also called a "is a" relationship:

A car *is a* vehicle

A student *is a* person

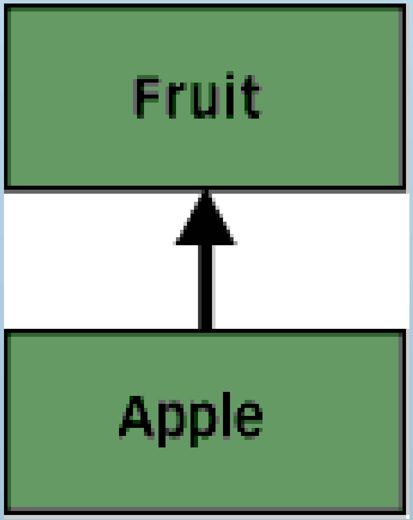
- Composition describes a "has a" relationship.
  One object is a part of another object.

A car *has* wheels

A person *has* legs

About Inheritance

```
class Fruit {
 // Return int number of pieces of peel that
  // resulted from the peeling activity.
    public int peel() {
  System.out.println("Peeling is appealing.");
        return 1;
    }
}
 class Apple extends Fruit {
}
 class Example1 {
 public static void main(String[] args) {
  Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```



Inheritance:

- When you run the Example1 application, it will print out "Peeling is appealing.", because Apple inherits (reuses) Fruit's implementation of peel().
- **If** at some point in the future, however, **you wish to change the return value of peel() to type Peel**, you will break the code for Example1.
- Your change to Fruit breaks **Example1's** code even though Example1 uses Apple directly and never explicitly mentions Fruit.
- For example, **if you change the return type of a public method in class Fruit** (a part of Fruit's interface), you can break the code that invokes that method on any reference of type Fruit or any subclass of Fruit.
- In addition, you break the code that defines any subclass of Fruit that overrides the method. Such subclasses won't compile until you go and change the return value of the overridden method to match the changed method in superclass Fruit

Inheritance…

```
class Peel
{
    private int peelCount;
    public Peel(int peelCount) {
        this.peelCount = peelCount;
    }
    public int getPeelCount() {
        return peelCount;
    }
    //...
}
class Fruit {
  // Return a Peel object that
    // results from the peeling activity.
    public Peel peel() {
        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}
```

```
// Apple still compiles and works fine
class Apple extends Fruit {
}
// This old implementation of Example1
// is broken and won't compile.

class Example1 {
    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

The composition alternative

- Given that the inheritance relationship makes it hard to change the interface of a superclass, it is worth looking at an *alternative approach provided by* Composition.
  - It turns out that when your goal is code reuse, composition provides an approach that yields easier-to-change code.
  - What is Composition, can we compare it with inheritance?

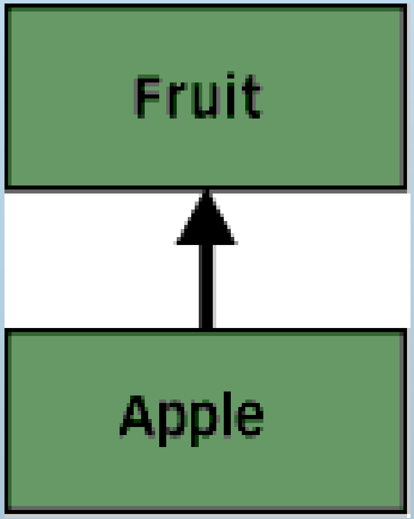
Composition

- Object composition is a way to combine simple objects or data types into more complex ones.
- A real-world example of composition may be seen in an automobile:
  - the objects *wheel*, *steering wheel*, *seat*, *gearbox* and *engine* may have no functionality by themselves, but an object called *automobile* containing all of those objects would serve a higher function, greater than the sum of its parts.
- A class can have references to objects of other classes as members.
- Composited (composed) objects are often referred to as having a "has a" relationship.

About Inheritance

- Here We'll be talking about single inheritance through class extension, as in:

```
class Fruit {

    //...

}

    class Apple extends Fruit {

    //...

}
```

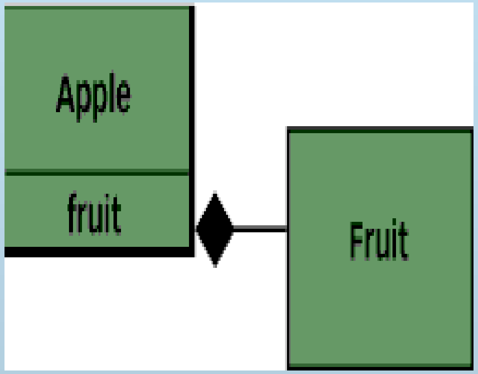


- In this simple example, ***class Apple is related to class Fruit by inheritance***, because Apple extends Fruit.

About Composition

We simply mean using ***instance variables*** that are references to other objects. For example:

```
class Fruit {
   //...
}
class Apple {
   private Fruit fruit = new Fruit();
   //...
}
```



➢**In the example above, class Apple is related to class Fruit by Composition, because Apple has an instance variable fruit that holds a reference to a Fruit object.**

➢**In this example, Apple is what we will call the *Front-end class* and Fruit is what we will call the *Back-end class*.**

➢**In a composition relationship, the Front-end class *holds a reference in one of its instance variables to a* Back-end class.**

```
class Engine
 {
 }
 class Automobile
 {
}
 class Car extends Automobile // car "is a" automobile //inheritance
    here
 {
 Engine engine; // car "has a" engine //composition here }
```

- **Composition** - Functionality of an object is made up of an aggregate of different classes. In practice, this means holding a pointer to another class to which work is deferred.
- **Inheritance** - Functionality of an object is made up of it's own functionality plus functionality from its parent classes.

- **Example**

- Construct a similar example of composition for the example of Automobile composed of *wheel*, *steering wheel*, *seat*, *gearbox* and *engine*

Composition

- In programming languages, composite objects are usually expressed by means of references from one object to another;
  - depending on the language, such references may be known as fields, members, properties or attributes, and the resulting composition as a structure, storage record, tuple, user-defined type (UDT), or composite type.
- However, having such references doesn't necessarily mean that an object is a composite.
- It is only called composite if the objects it refers to are really its parts, i.e. have no independent existence.

Advantages/Disadvantages Of Inheritance

- **Advantages:**

  - New implementation is easy, since most of it is inherited

  - Easy to modify or extend the implementation being reused

- **Disadvantages:**

  - **Breaks encapsulation**, since it exposes a subclass to implementation details of its superclass

  - "**White-box**" reuse, since internal details of superclasses are often visible to subclasses

  - Subclasses may have to be changed if the implementation of the superclass changes

  - Implementations inherited from superclasses **can not be changed at runtime**

The composition alternative

- Given that the inheritance relationship makes it hard to change the interface of a superclass, it is worth looking at an ***alternative approach provided by composition***.

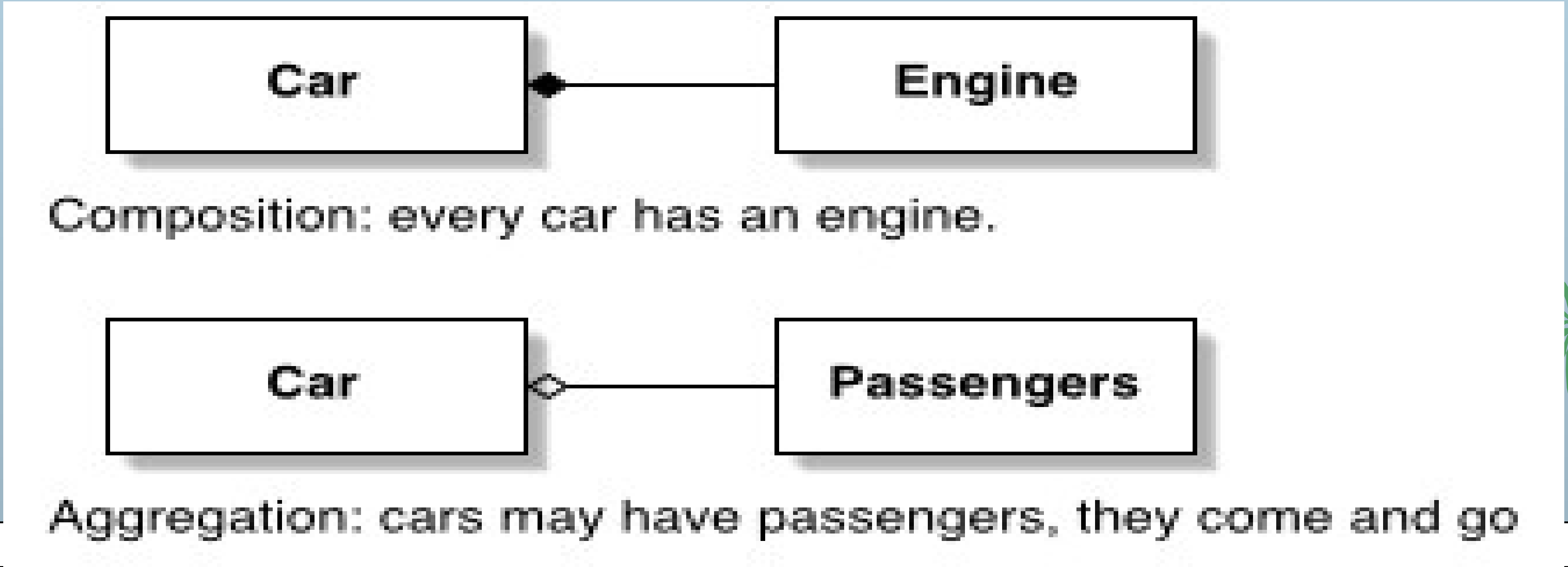
- It turns out that when your goal is code reuse, composition provides an approach that yields easier-to-change code.

Favor Composition Over Inheritance

- Method of reuse in which new functionality is obtained by creating an object *composed of* other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- Sometimes called ***aggregation*** or ***containment***, although some authors give special meanings to these terms

Aggregation

- Aggregation differs from ordinary composition in that it does *not imply ownership.*
  - In composition, **when the owning object is destroyed, so are the contained objects**. In aggregation, this is not necessarily true.



Composition: every car has an engine.

Aggregation: cars may have passengers, they come and go

Composition..revisited

- Object composition is a way to combine simple objects or data types into more complex ones.
- Compositions are a critical building block of many basic data structures, including the *tagged union, the linked list, and the binary tree*, as well as the object used in object-oriented programming.
- Composited (composed) objects are often referred to as having a "has a" relationship.
- A real-world example of composition may be seen in an automobile:
  - the objects ***wheel***, ***steering wheel***, ***seat***, ***gearbox*** and ***engine*** may have no functionality by themselves, but an object called ***automobile*** containing all of those objects would serve a higher function, greater than the sum of its parts.

Aggregation

- Aggregation differs from ordinary composition in that it does not imply ownership.
- In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true.
    - For example, a university owns various departments (e.g., chemistry), and each department has a number of professors.
    - If the university closes, the departments will no longer exist, but the professors in those departments will continue to exist.
- Therefore, a University can be seen as a composition of departments, whereas departments have an aggregation of professors.
- In addition, a *Professor could work in more than one department, but a department could not be part of more than one university*.

Advantages/Disadvantages Of Composition

- **Advantages:**

  - Contained objects are accessed by the containing class solely through their interfaces

  - "Black-box" reuse, since internal details of contained objects are *not* visible

  - Good encapsulation

  - Fewer implementation dependencies

  - Each class is focused on just one task

  - The composition can be defined dynamically at run-time through objects acquiring references to other objects of the same type

- **Disadvantages:**

  - Resulting systems tend to have more objects

  - Interfaces must be carefully defined in order to use many different objects as composition blocks

For example:

- Aggregation - when one object owns or is responsible for another object and both objects have identical lifetimes (**GoF**)
- Aggregation - when one object has a collection of objects that can exist on their own (UML)
- **Aggregation is a special form of association. It is also a relationship between two classes like association, however its a directional association, which means it is strictly a one way association. It represents a Has-A relationship.**
- Containment - a special kind of composition in which the contained object is hidden from other objects and access to the contained object is only via the container object

Inner Class
Design Patterns