GRAND VALLEY STATE UNIVERSITY
www.gvsu.edu

Software Engineering
OOP, OOD, OOA

Erik Fredericks // frederer@gvsu.edu

Adapted from materials provided by Byron DeVries, Jagadeesh Nandigam

OOSD

Object Oriented (OO) Software Development consists of three main components:

- Object-Oriented Analysis (OOA)
- Object-Oriented Design (OOD)
- Object-Oriented Programming (OOP)

What's the difference?

Object-oriented analysis

Object-Oriented **Analysis** (OOA) is concerned with:

- Developing an object-oriented model of the application domain.
- Identification of objects/entities and operations associated with the problem.

Object-oriented design

Object-Oriented **Design** (OOD) is concerned with:

- Developing an object-oriented model of the system to implement **requirements**

- **Implementing** the solution by **adding new objects** to the ones already identified on the OOA phase

Object-oriented

Analysis

and

Design

Do the right thing

Analysis emphasizes an investigation of the problem and requirements, rather than a solution.

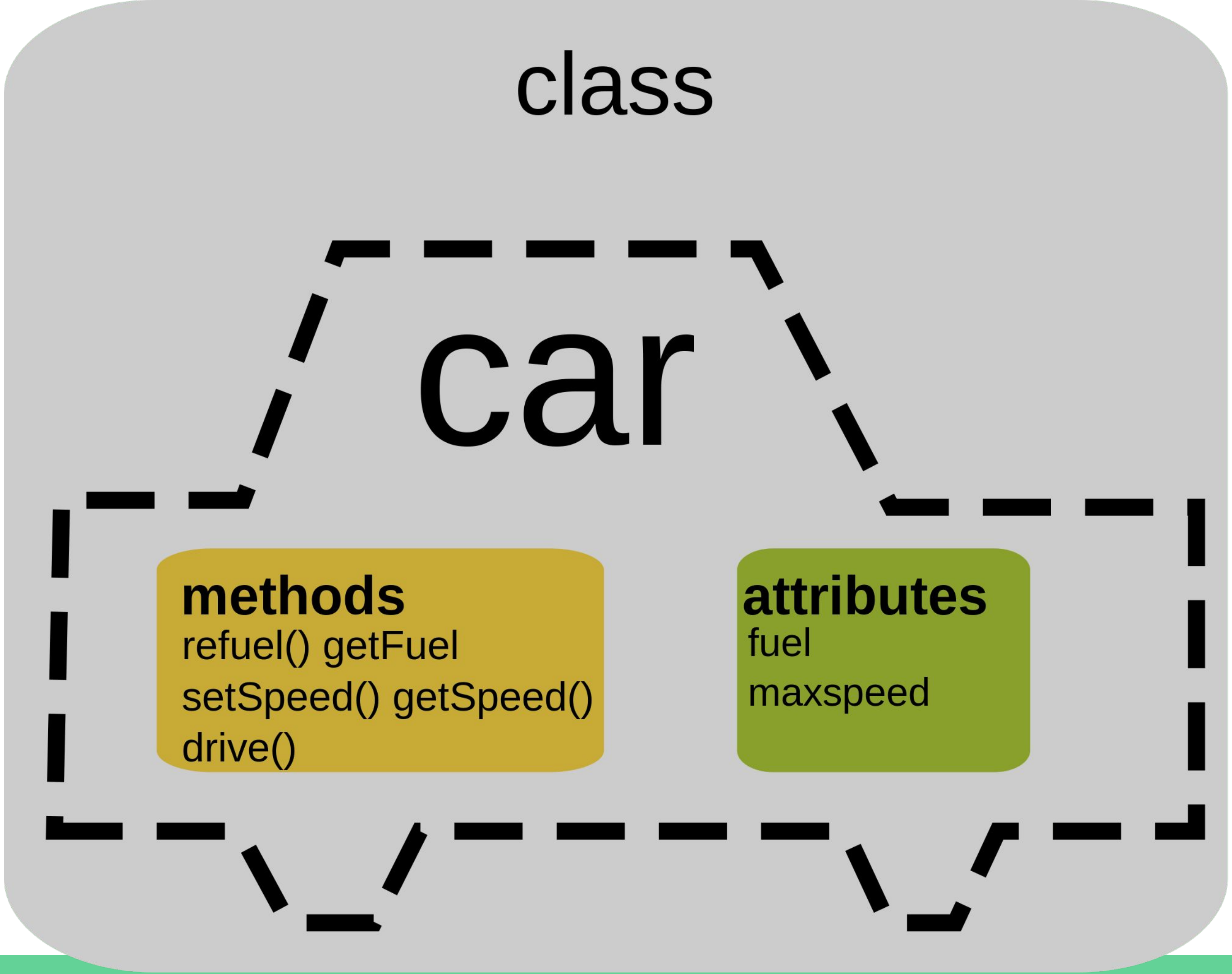
Do the thing right

Design emphasizes a conceptual solution that fulfils the requirements, rather than its implementation.

OOP (there it is)

Object-Oriented **Programming** (OOP) is concerned with:

- **Realizing an OOD** using an object-oriented programming language.
- Identifying **additional objects that are language or API specific** and necessary to implement the solution.


class
car
methods
refuel() getFuel
setSpeed() getSpeed()
drive()
attributes
fuel
maxspeed

OO design principles

- Encapsulation
- Information Hiding
- Object Reuse
  - Inheritance vs composition
  - Delegation with composition
  - Favor object composition over class inheritance

- Object Interfaces
- Program to an *interface*, not an implementation

Encapsulation

Encapsulation: Bundling of data with operations that operate on that data.

Operations are:

- Methods in OO programming languages
- Functions and procedures in procedural programming languages

Encapsulation is a **language facility / construct**.

Encapsulation **is not specific to just OO** programming languages

Encapsulation

Encapsulation can be achieved in programming languages that provide a facility to **bundle data and operations** together.

- In Ada, encapsulation is achieved using the package construct
- In Modula, encapsulation is achieved using the module construct
- In ML, it is achieved using the abstype construct
- In C, it is achieved via pointer trickery to mimic a map
    - https://alastairs-place.net/blog/2013/06/03/encapsulation-in-c/

Encapsulation is **not the same** as information hiding.
How could you have data that is not hidden, but is encapsulated in Java?

class

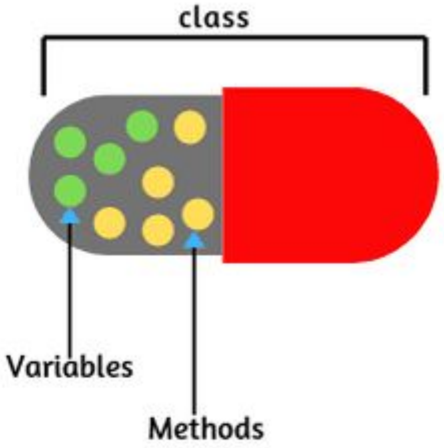
{

data members

+

methods (behavior)

}



Information hiding : BACKGROUND

Concept first introduced by David Parnas in **1972**.

Design principle that strives to shield client modules from the internal works of a module.

Parnas stressed hiding ”difficult design decisions or design decisions which are **likely to change**”

Encapsulation *facilitates*, but does not guarantee, information hiding.

Information hiding: RULES

Don’t expose data

- Make all data items private and use operations (getters and setters?)
- Makes defensive programming possible

Don’t expose the difference between stored data and derived data

- Whether a data value is stored or derived is a design decision best kept hidden.
- Example: Use an accessor method named `speed()` or `getSpeed()` rather than `calculateSpeed()` to return an attribute called `speed` of an object

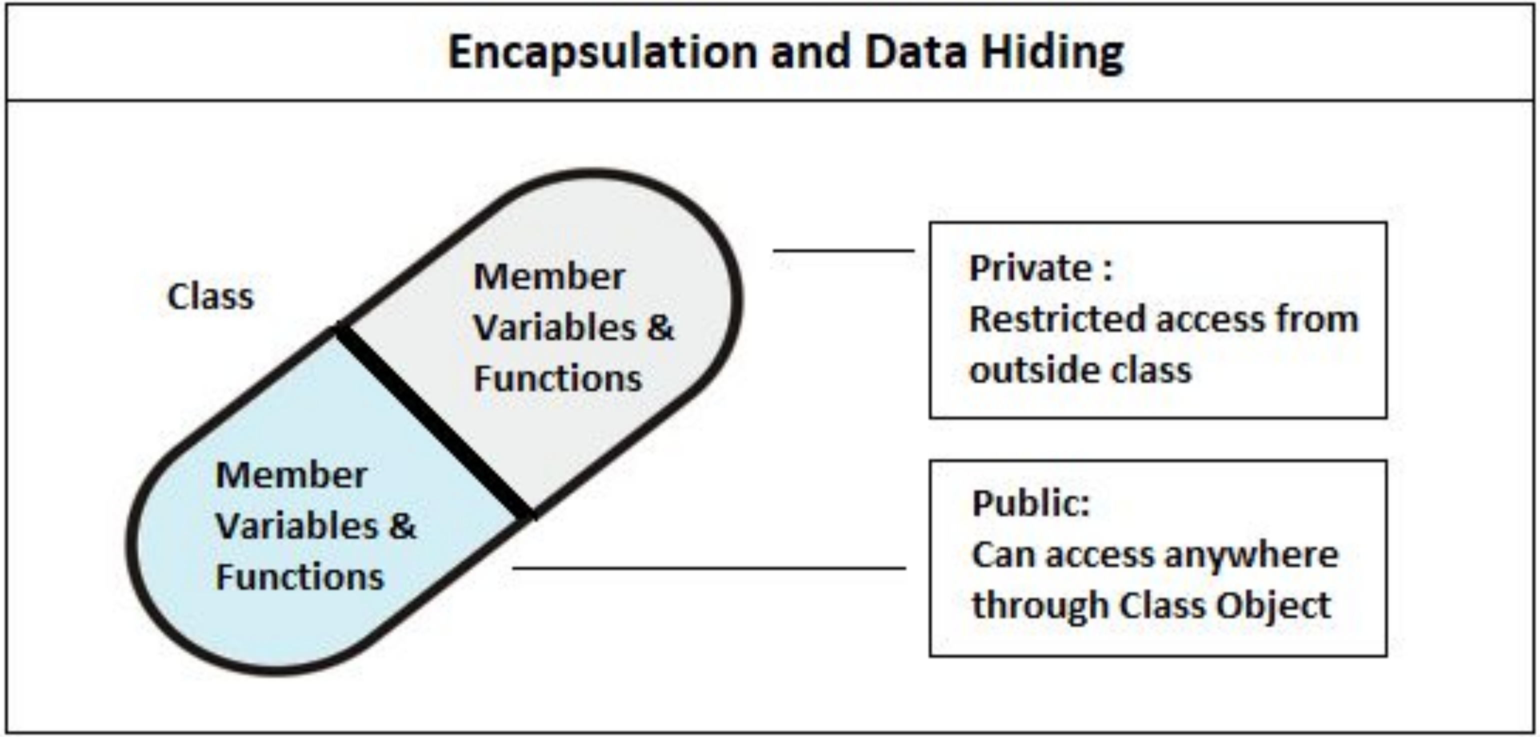
Information hiding : MORE RULES

Don’t expose a class’s internal structure

- Clients should remain isolated from the design decisions driving the selection of internal class structure

Don’t expose implementation details of a class

Remember: Encapsulation is not information hiding.


Encapsulation and Data Hiding
Class
Member
Variables &
Functions
Member
Variables &
Functions
Private :
Restricted access from
outside class
Public:
Can access anywhere
through Class Object

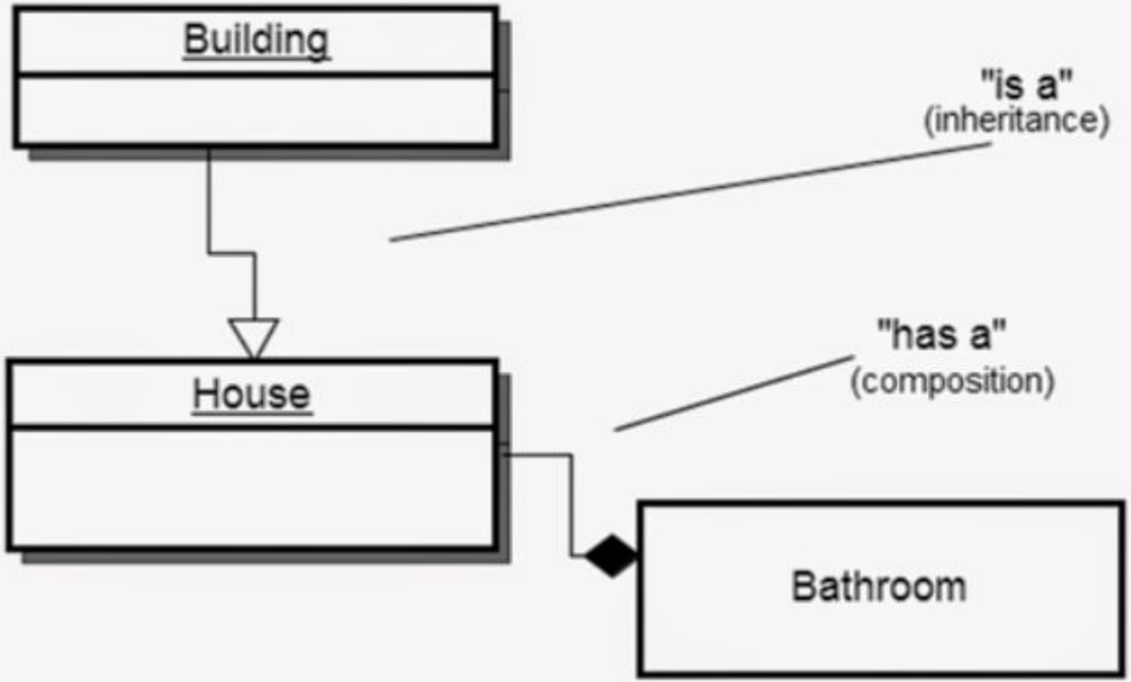
Reuse

Two common techniques for reusing functionality in object-oriented systems are:
- Class inheritance
- Object composition

Class Inheritance:
- Reuse by sub-classing / extending / inheriting
- Often referred to as **white-box reuse**

Object Composition:
- New functionality is obtained by creating an object composed of other objects.
- Also known as **black-box reuse**

Composition vs. Inheritance



Reuse with **inheritance**

A method of reuse in which new functionality is obtained by extending the implementation of an existing object.

The **generalization class** (*superclass*) explicitly captures the common attributes and methods.

The **specialization class** (*subclass*) extends the implementation with additional attributes and methods.

Advantages:

- New implementation is easy, since most of it is inherited
- Easy to modify or extend the implementation being reused

Any disadvantages?

Reuse with inheritance

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** (i.e., class inheritance is defined statically at compile-time)

Reuse with **composition**

New functionality is obtained by **delegating functionality** to one of the objects being composed.

Object composition is **defined dynamically at run-time** through objects acquiring references to other objects.

Reuse with **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 (due to information hiding)

Disadvantages:

- Resulting systems tend to have more objects
- Interfaces must be carefully defined in order to use many different objects as composition blocks.

Object interfaces

An interface is a **set of methods** one object knows it can invoke on another object.

An object can have many interfaces

A type is a specific interface of an object.

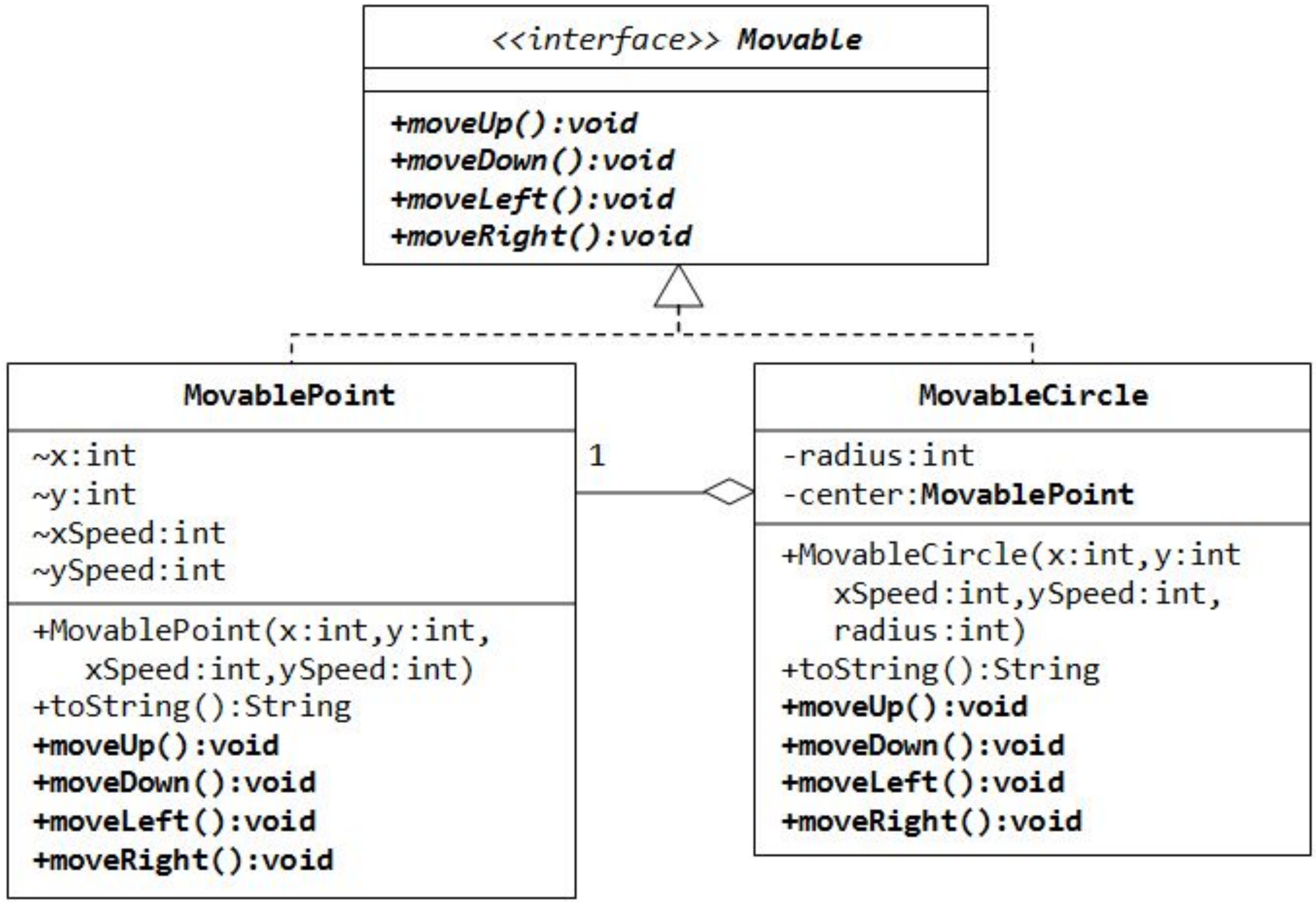
Different objects can have the same type and the same object can have many different types.

An object is known by other objects only through its interface.

Interfaces are the key to **pluggability**

A principal of reusable object-oriented design:

“Program to an interface, not an implementation”


<<interface>> Movable
+moveUp():void
+moveDown():void
+moveLeft():void
+moveRight():void
MovablePoint
~x:int
~y:int
~xSpeed:int
~ySpeed:int
+MovablePoint(x:int,y:int,
xSpeed:int,ySpeed:int)
+toString():String
+moveUp():void
+moveDown():void
+moveLeft():void
+moveRight():void
1
MovableCircle
-radius:int
-center:MovablePoint
+MovableCircle(x:int,y:int
xSpeed:int,ySpeed:int,
radius:int)
+toString():String
+moveUp():void
+moveDown():void
+moveLeft():void
+moveRight():void

Object interfaces

Advantages:

- Clients are unaware of the specific class of the object they are using
- One object can be easily replaced by another
- Object connections need not be hardwired to an object of a specific class (increased flexibility)
- Loosens coupling
- Increases likelihood of reuse
- Improves opportunities for composition since contained objects can be of any class that implements a specific interface

Disadvantages:

- Modest increase in design complexity

Interface example

```
/** Interface IManeuverable provides the specification
*   for a maneuverable vehicle.
*/
public interface IManeuverable {
    public void left();
    public void right();
    public void forward();
    public void reverse();
    public void climb();
    public void dive();
    public void setSpeed(double speed);
    public double getSpeed();
}
```

Interfaces: Example implementations

```
public class Car implements IManeuverable {
    // Code here.
}

public class Boat implements IManeuverable {
    // Code here.
}

public class Submarine implements IManeuverable {
    // Code here.
}
```

Interfaces: Example usage

The method below from some other class can maneuver the vehicle without being concerned about what the actual class is (Car, Boat, Submarine) or what inheritance hierarchy this other class is in.

```
public void travel(IManeuverable vehicle) {
    vehicle.setSpeed(35.0);
    vehicle.forward();
    vehicle.left();
    vehicle.climb();
}
```

Python example

https://realpython.com/python-interface/

https://www.geeksforgeeks.org/python-interface-module/

http://masnun.rocks/2017/04/15/interfaces-in-python-protocols-and-abcs/

Assessing design **quality**

“A module is a lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier” (Yourdon & Constantine)

By this definition, a module can be a:

- Function, procedure, or method
- Class

Two software quality metrics were proposed (in the early 1970s by Stevens, Myers, and Constantine) for judging the quality of a module:

- Module **cohesion**
- Module **coupling**

Module **cohesion**

Cohesion is the degree of relatedness/similarity between elements within a module. It is an intra-module measure.
Levels of Cohesion:

- Functional Cohesion (Best / Most Desirable)
- Sequential Cohesion
- Communicational Cohesion
- Procedural Cohesion
- Temporal Cohesion
- Logical Cohesion
- Coincidental Cohesion (Worst / Least Desirable)

Modules with high cohesion tend to be more maintainable (i.e., modifiable and understandable) and reusable compared to modules with low cohesion.

How do you improve the cohesion of a module?

How would you improve module cohesion?

Cohesion

Functional Cohesion (Best / Most Desirable)

- Parts grouped to do one well-defined task

Sequential Cohesion

- Output of one is input to another

Communicational Cohesion

- Operate on same data

Procedural Cohesion

- Grouped because they follow a particular sequence of operations

Temporal Cohesion

- Parts processed at particular time

Logical Cohesion

- Grouped by logical categories (but differ in nature)

Coincidental Cohesion (Worst / Least Desirable)

- Parts of module grouped arbitrarily

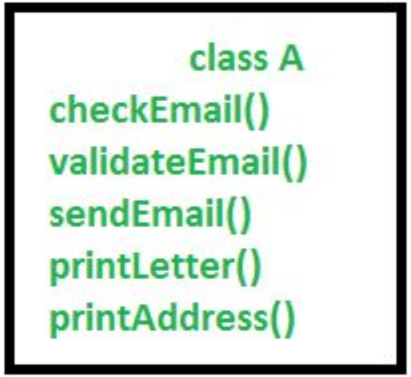


Fig: Low cohesion

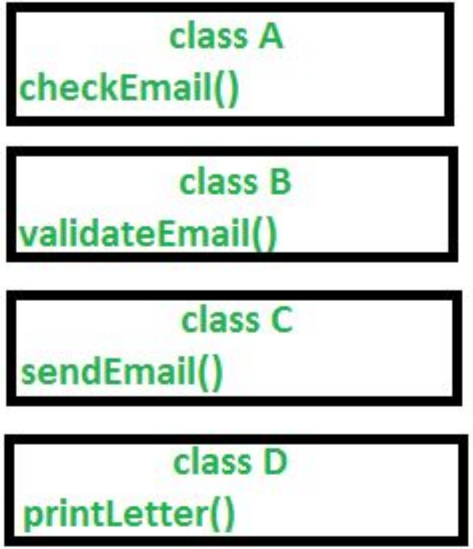


Fig: High cohesion

Module **coupling**

Coupling is the degree of **dependence** between two or more modules.
Types of coupling:

- Normal Coupling (Low / Most Desirable)
  - Data
  - Control
- Common Coupling
- Content Coupling (Worst / Least Desirable)

Normal Coupling:

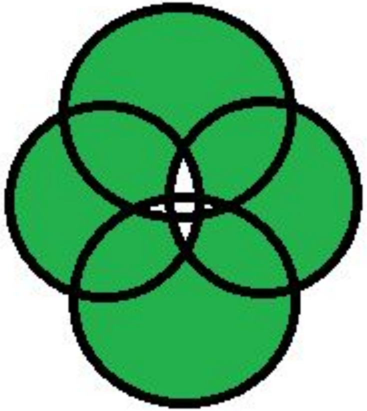
- Coupling by parameters / calls

Common Coupling:

- Coupling via global data

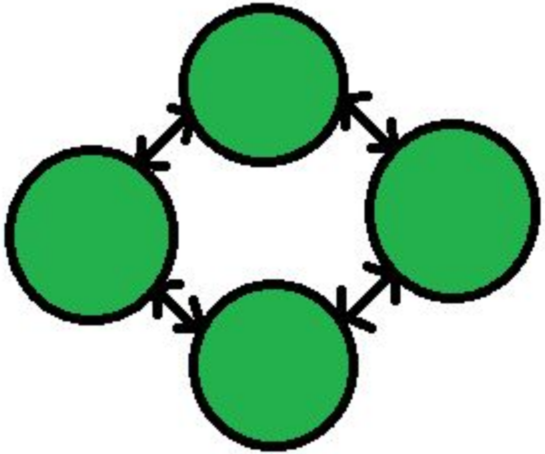
Content Coupling:

- Coupling that occurs when one module directly references or changes the contents (data or code) of another module.



Tight coupling:

1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:

1. Less Interdependency
2. Less coordination
3. Less information flow

What is the relationship between design patterns and OOP/OOD/OOA?

Is one better than the other?

Do you think design patterns informed OOP/OOD/OOA or OOP/OOD/OOA informed design patterns?

Is one better than the other (OOA/OOD/OOP)? Why?