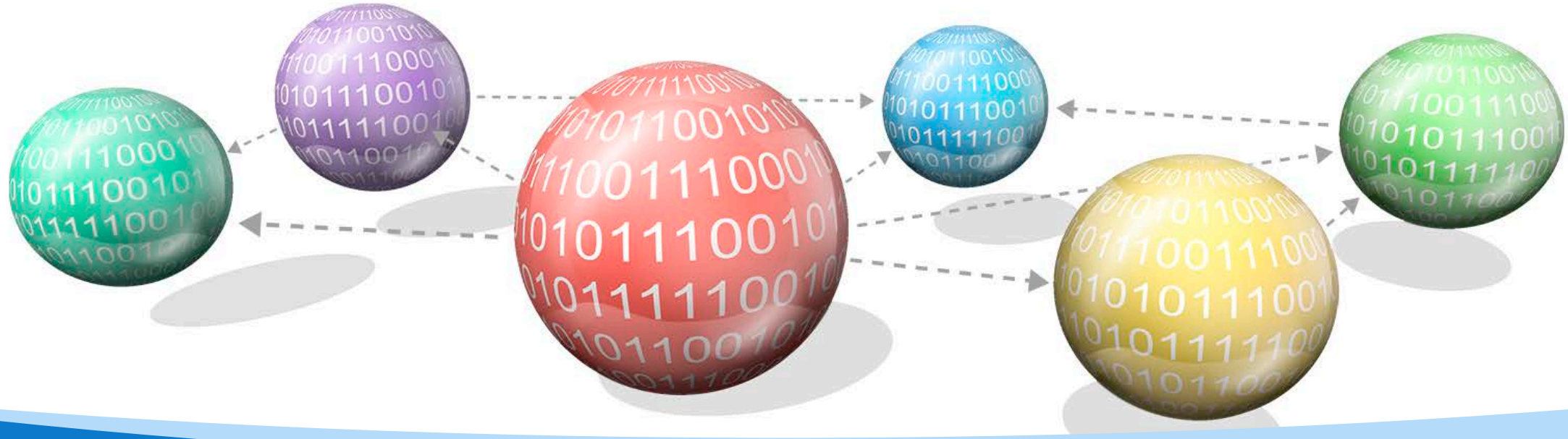


CE/CZ2002 Object-Oriented Design & Programming

Chapter 9: Design Principles

Mr Tan Kheng Leong
Lecturer, School of Computer Science and Engineering

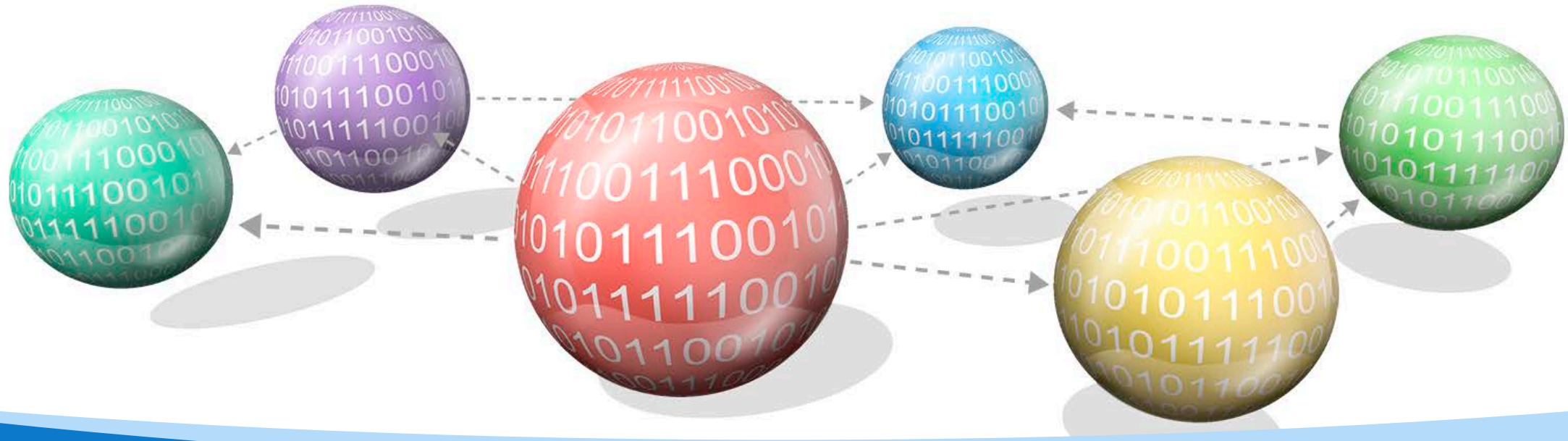
2021!
25% group project!
8% exam!

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Objectives

By the end of this chapter, you should be able to:

- Explain the characteristics of Good and Bad design
- Describe the various OO design goals
- Explain SOLID design principles



CE/CZ2002 Object-Oriented Design & Programming

Topic 1: Characteristics of Good and Bad Design

Chapter 9: Design Principles

Mr Tan Kheng Leong
Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

What is Good Design ?

- A working piece of software may not necessarily mean that it is a good designed software.
- It may be working at this instant, but inherently may show symptoms of breaking or design issues.

What is Bad Design ?

Design Fails. Retrieved December 16, 2016 from http://designonthesquare.com/architectblog/wp-content/uploads/sites/4/2010/06/faucet.jpg.

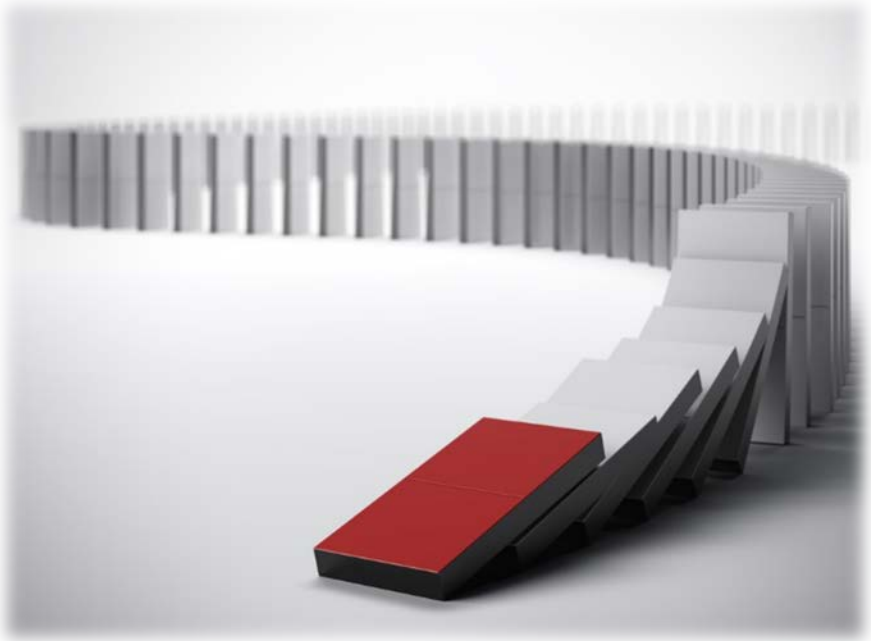
Diy Home Improvement Funny Fails. Retrieved December 16, 2016 from http://www.donutrockcity.com/storage/static/img/content/diy-home-improvement-funny-fails.jpeg.

Symptoms of Rotting Design

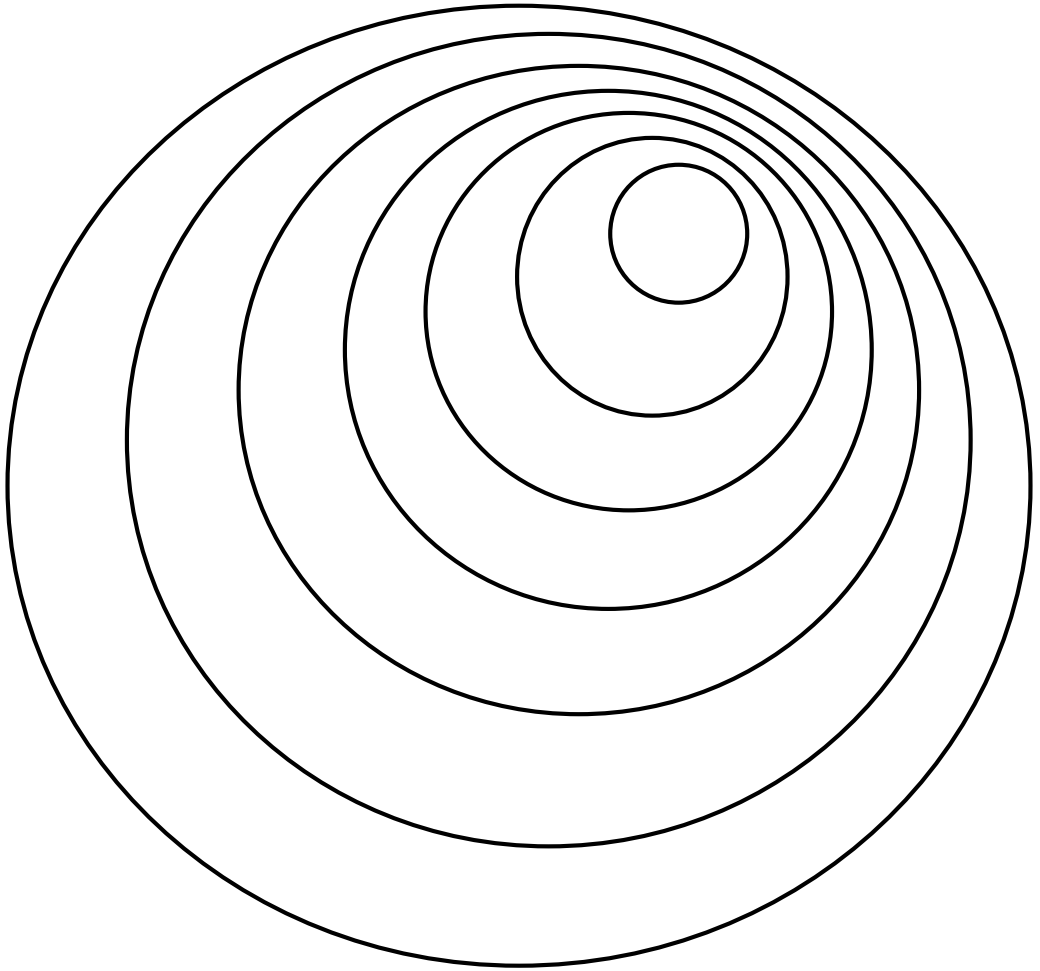
- **Rigidity** (One change lead to another change)
  - The tendency for software to be difficult to change, even in simple ways.
  - Every change causes a cascade of subsequent changes in dependent modules.
- **Fragility**
  - The tendency of the software to break in many places every time it is changed.
  - Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.
- **Immobility**
  - The inability to reuse software/module from other projects or from parts of the same project.
  - The module in question has too much baggage it depends upon.

Rigidity

Change Cascades

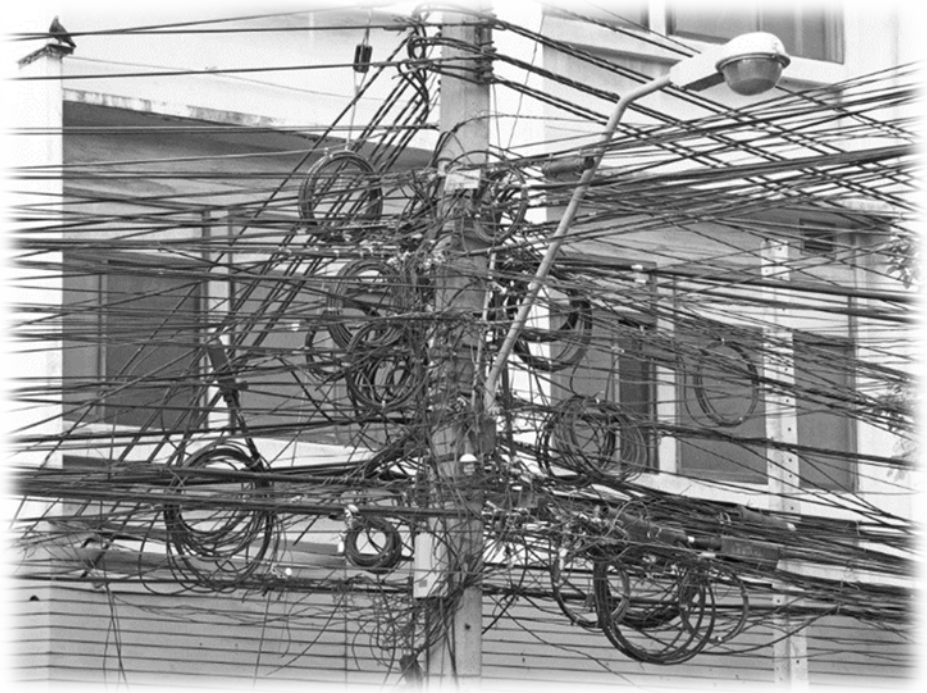


Ripple /Snowball Effect

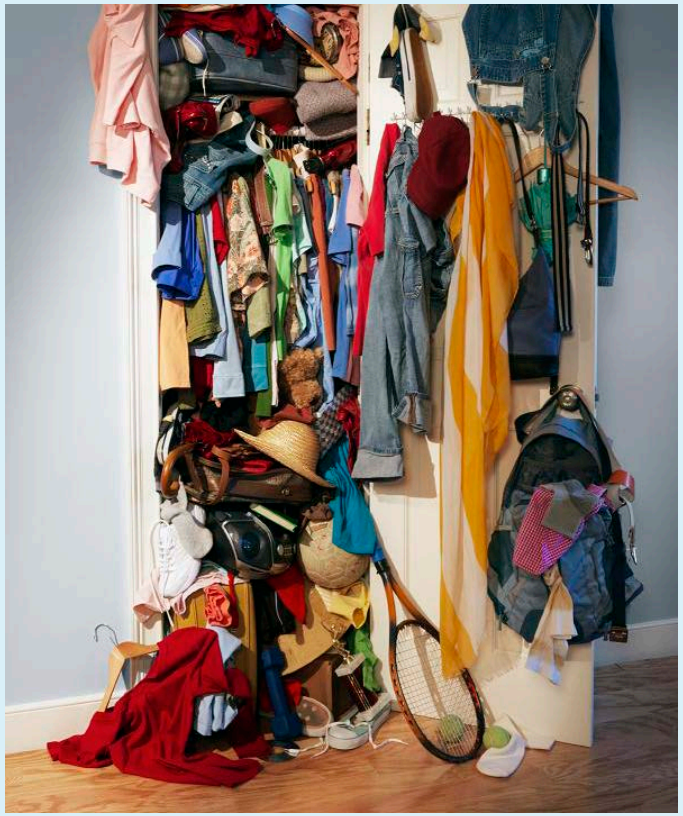


Immobility

Try unplug a cable?



Fragility



Stress Reliever-My Favorite Junk Shop. Retrieved May 29, 2017 from http://petticoatjunktion.com/wp-content/uploads/2013/09/rusty-treasures.jpg.

Wardrobe Solutions. Retrieved May 29, 2017 from http://blog.trendin.com/wp-content/uploads/2014/12/Trendin_wardrobe-solutions_messy.jpg

Good Design and Programming

… is not learned by generalities, but by seeing how significant programs can be made clean, easy to read, easy to maintain and modify, human-engineered, efficient, reliable, and secure, by the application of good design and programming practices.

Careful study and imitation of good designs and programs significantly improves development skills.

Pro (efficient) Con (Readability)

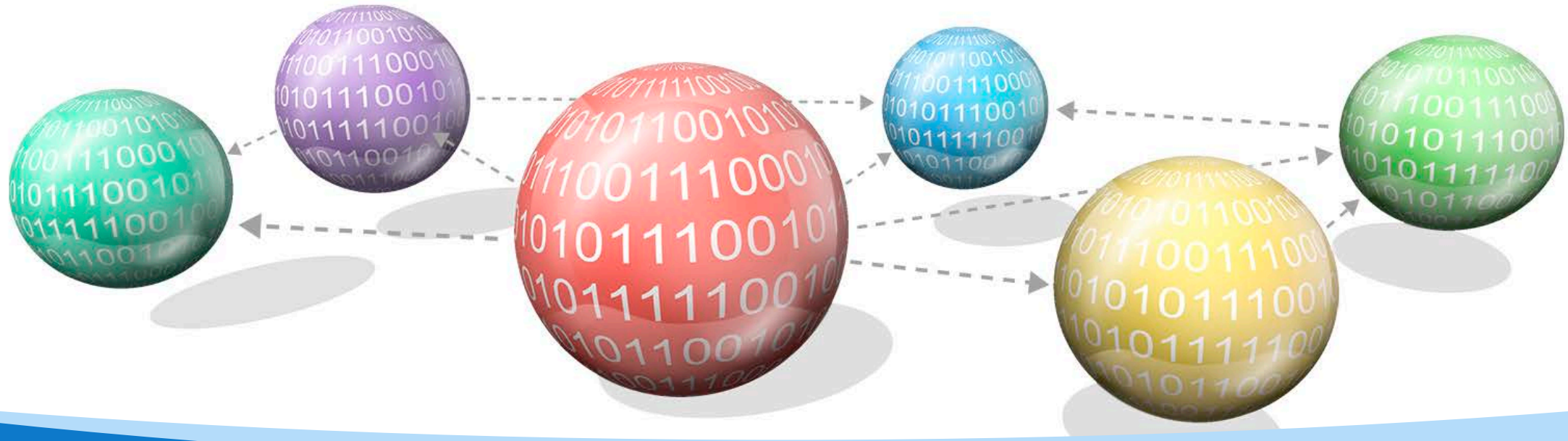
Method chaining

```
order.getCustomers(criteria).iterator().next().getName().getFirstName().get(0);
```

```
sc.next().charAt(0);
```

Kernighan and Plauger

Authors of 'The *Elements of Programming Styles*'



CE/CZ2002 Object-Oriented Design & Programming

Topic 2: Object Oriented Design Goals

Chapter 9: Design Principles

Mr Tan Kheng Leong
Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

OO Design Goals

Make software **easier to change** when we want to.

- We might want to change **a class or package** to add new functionality, change business rules or improve the design.
- We might have to change a class or package because of a change to **another class or package it depends on** (e.g., a change to a method signature).
- **Manage dependencies** between classes and packages of classes to **minimise impact** of change on other parts of the software.
- Minimise reasons that modules or packages might be forced to change because of a change in a module or package it depends upon.

OO Design Goals

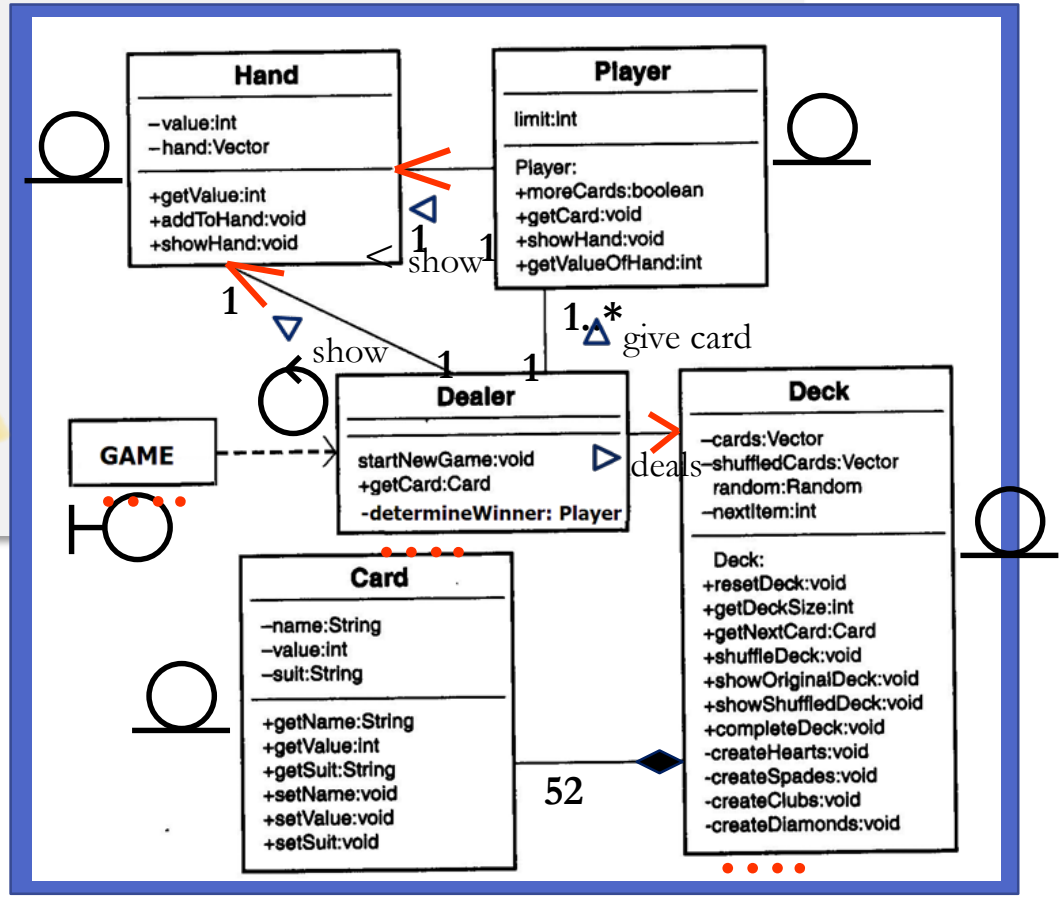
- Much of OO design is about **managing dependencies.**
- It is very difficult to write OO code without creating a dependency on something.

Class Design Guidelines

Manage

- Coupling and Cohesion
  - Design with **Reuse** (Reusability) in mind
  - Design with **Extensibility** in mind
  - Design with **Maintainability** in mind
- To achieve

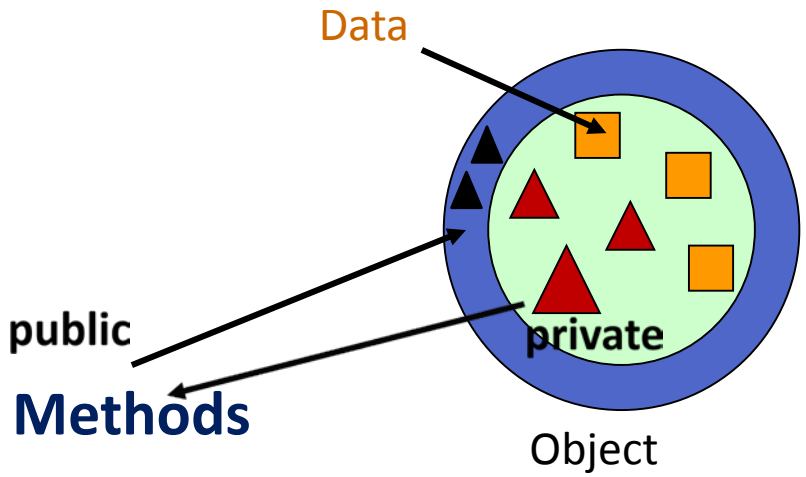
  Loose (Low) **Coupling** and High **Cohesion**



Designing for Change

A modular program has **well-defined**, conceptually **simple** and **independent** units interacting through well-defined **interfaces.**

- Encapsulation

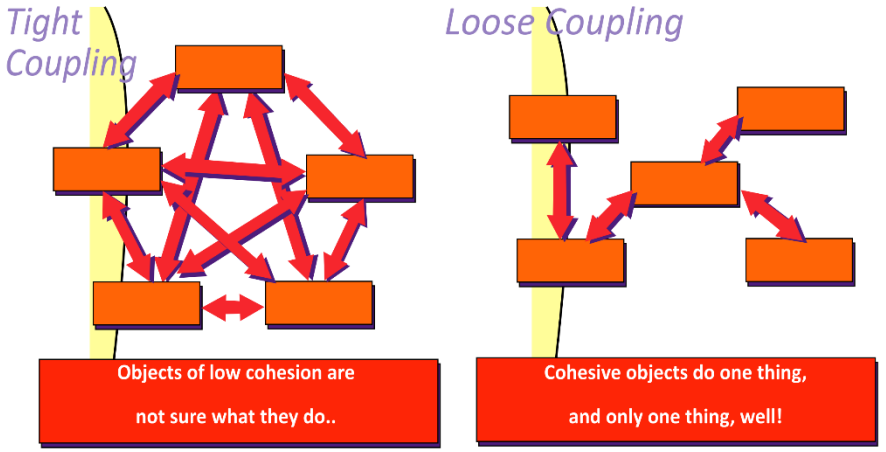


Designing for Change

A modular program has **well-defined**, conceptually **simple** and **independent** units interacting through well-defined **interfaces.**

- Encapsulation
- ~~Tight~~/Loose Low Coupling (linking classes tgt)

Class Design ≠ Network Design



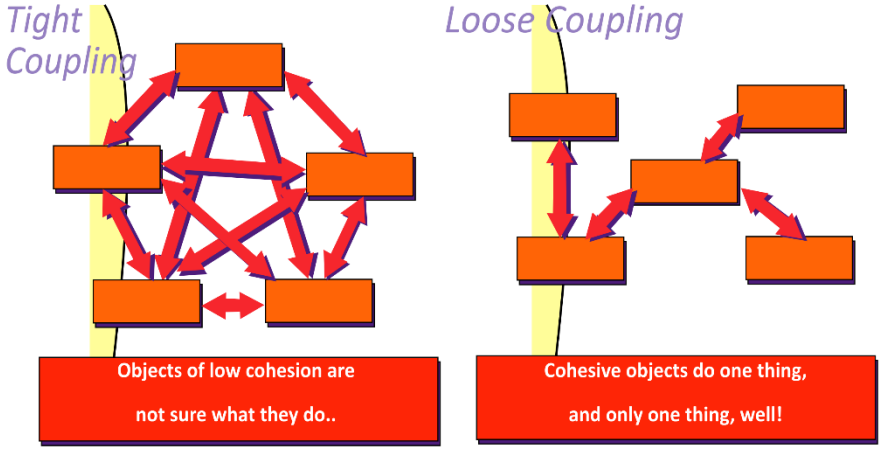
Designing for Change

A modular program has **well-defined**, conceptually **simple** and **independent** units interacting through well-defined **interfaces.**

- Encapsulation
- ~~Tight~~/Loose
- Low Coupling
- High Cohesion

Only link when needed link most node

Class Design ≠ Network Design



Example – Data Persistence

Refer to video at: 07:41

DataManager
validateCustomer(..):Boolean storeCustomer(..):Boolean validateAccount(..):Boolean storeAccount(..):Boolean

IF Pass Chng then Datamanager need to change

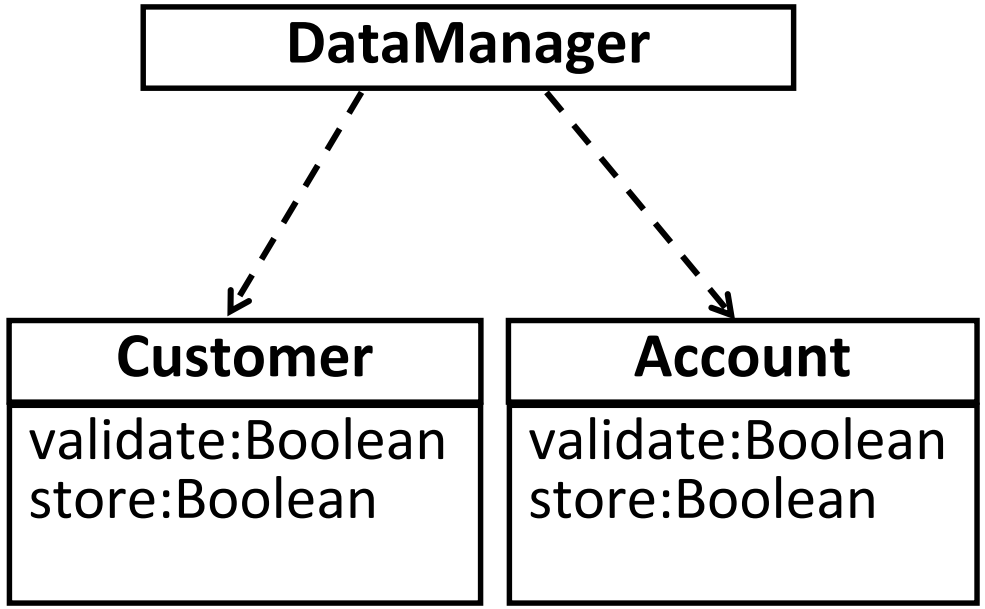
```
class DataManager {
...
    boolean  validateCustomer(String cusID,
String pwd) {
        String passwd = ...../// get from DB the
            //customer password that match cusID
        if (passwd().equals(pwd))
            return true;
        return false ; }
.....
}
```

Bad Encapsulation – DataManager needs to know about the inner working of Customer and Account

Example – Data Persistence

Solution

```
class DataManager {
...
     boolean  validateCustomer(String cusID, String pwd)
{
          Customer c = new Customer(cusID)
               return c.validate(String pwd);
     }
.........
}

class  Customer {
......
     boolean validate(String pwd) {
        // compare plain pwd here
        // Or hash  Or .. …
}
```



Example – Data Persistence

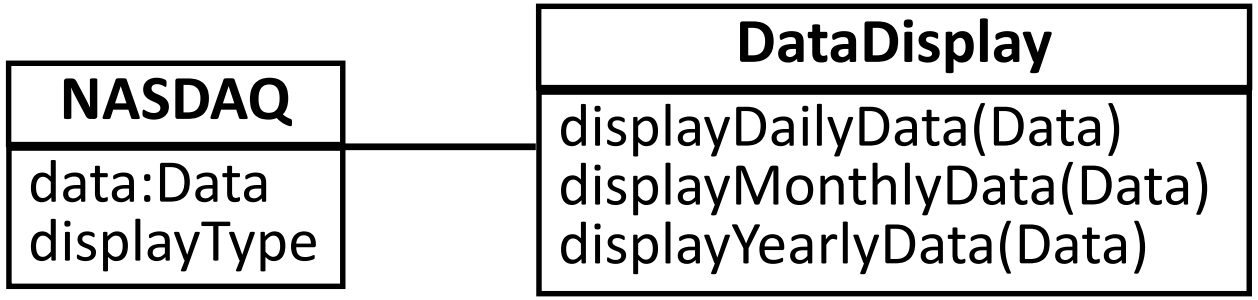
```
class DataManager {
...
    boolean  validateCustomer(String cusID, String pwd)
{
        Customer c = new Customer(cusID)
            return c.validate(String pwd);
    }
.....
}

class  Customer {
……
    boolean validate(String pwd) {
       // compare plain pwd here
       // Or hash  Or .. …
}
```

```
class DataManager {
...
    boolean  validateCustomer(String cusID,
String pwd) {
        String passwd = ...../// get from DB the
            //customer password that match cusID
        if (passwd().equals(pwd))
            return true;
        return false ; }
.....
}
```

Better Encapsulation – Customer and Account responsible for own validation and persistence, DataManager manages the entities.

Example – Finance Index Display

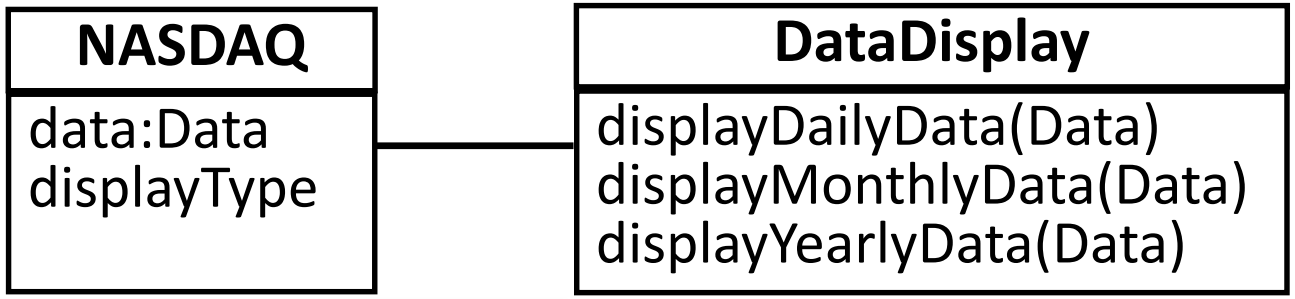
Refer to video at: 10:46



- Tight Coupling – NASDAQ and DataDisplay need each other to work.
- Low Cohesion – NASDAQ needs to track data and manage display.
- Bad Encapsulation – NASDAQ needs to know inner workings of DataDisplay, i.e. the displayType.

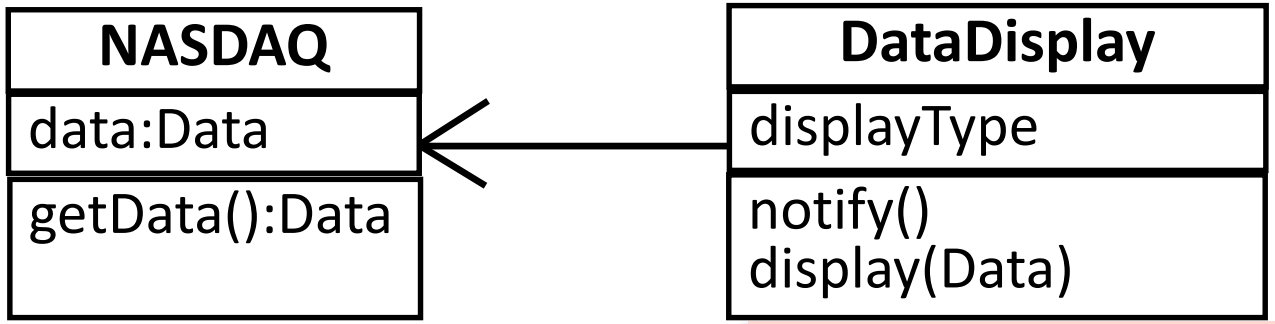
Nasdaq Composite. Retrieved May 29, 2017 from https://finance.yahoo.com/quote/%5EIXIC?p=%5EIXIC.

Example – Finance Index Display



- let data display do display
- NASDAQ just provide data

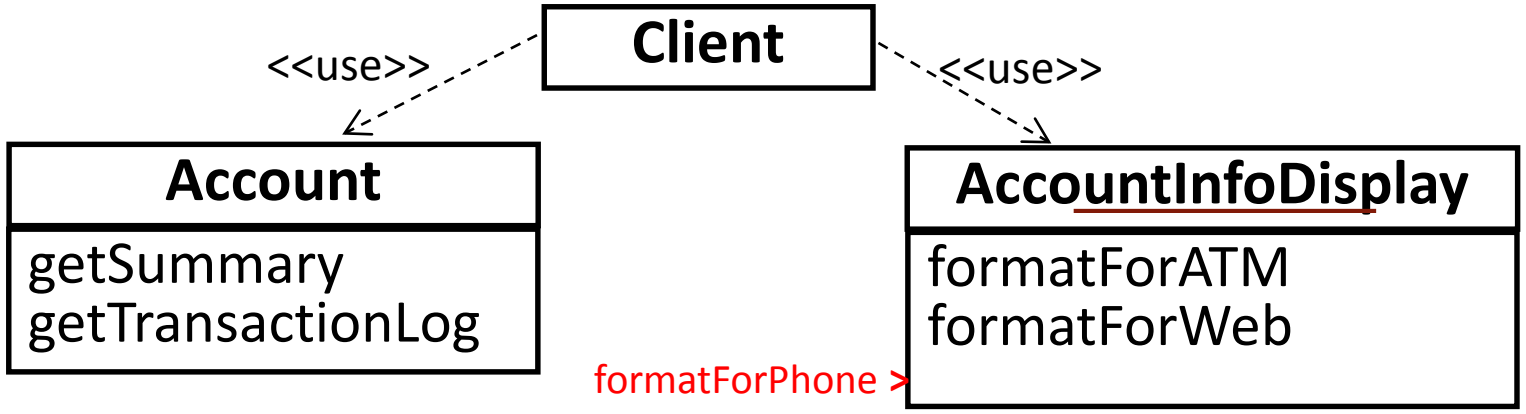
```
if (displayType == DAILY)
      dataDisplay.displayDailyData(getData())
else (displayType == MONTHLY) …….
```



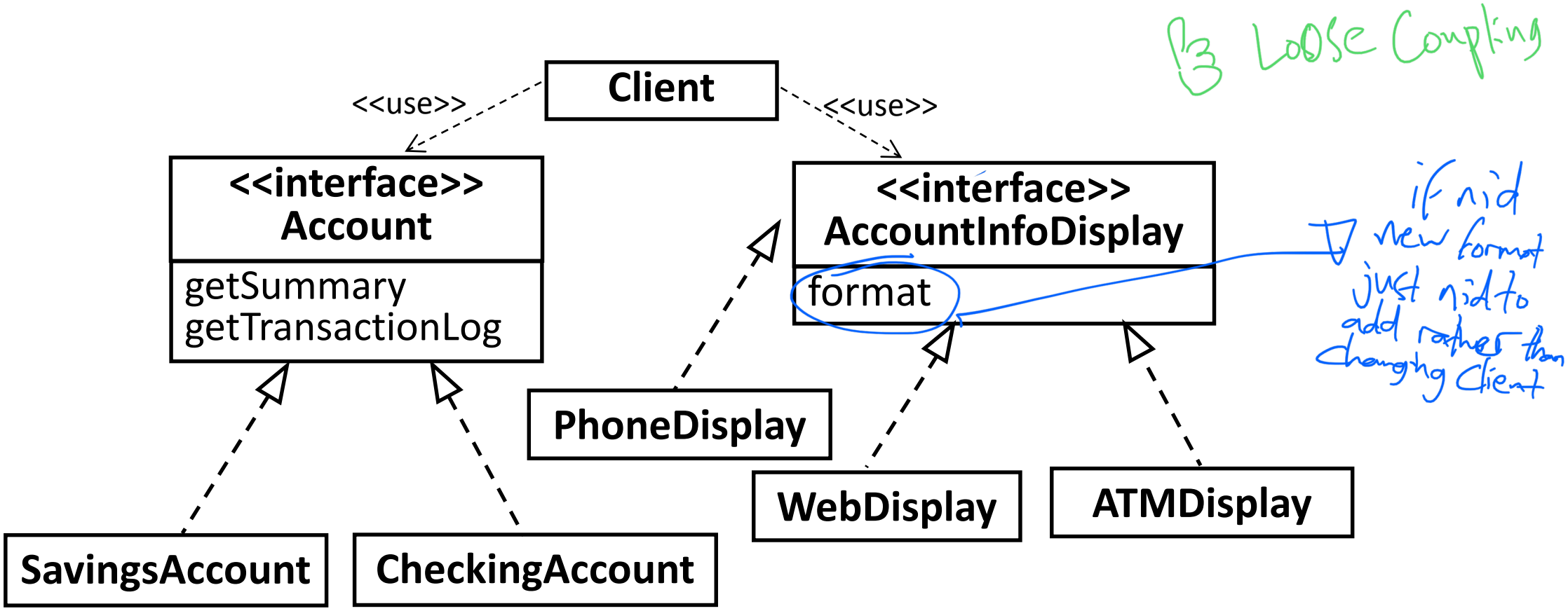

```
public void notify()   {  display(NASDAQ.getData()  }
private void display(Data  d){
      if (displayType == DAILY)
            // display daily chart
      else …..
```

Example – Bank Account

Refer to video at: 13:31



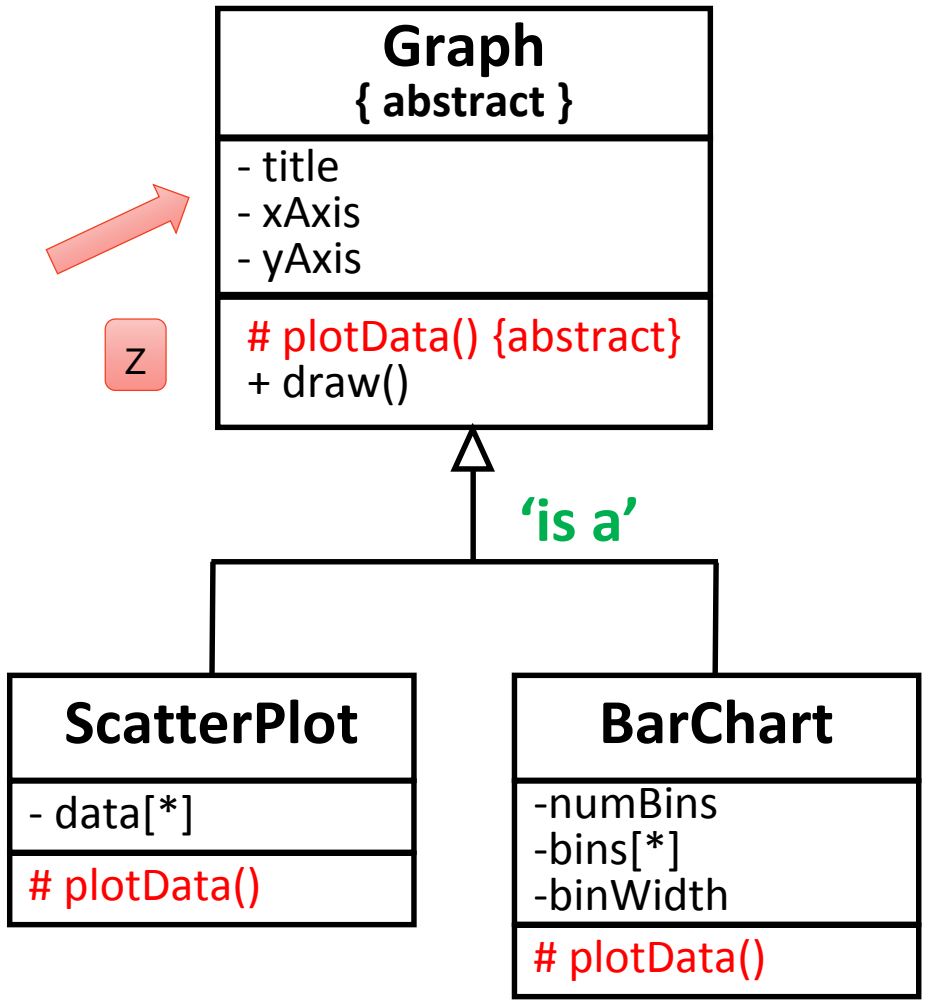
Example – Bank Account

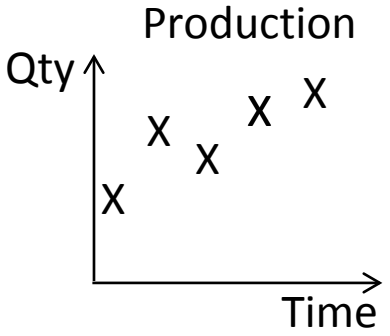


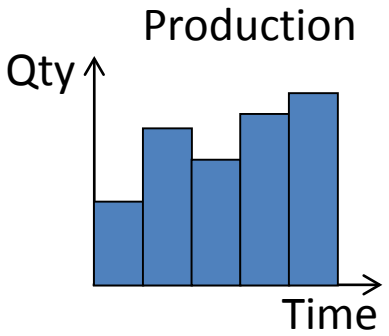
Interaction via interfaces promotes loose coupling.

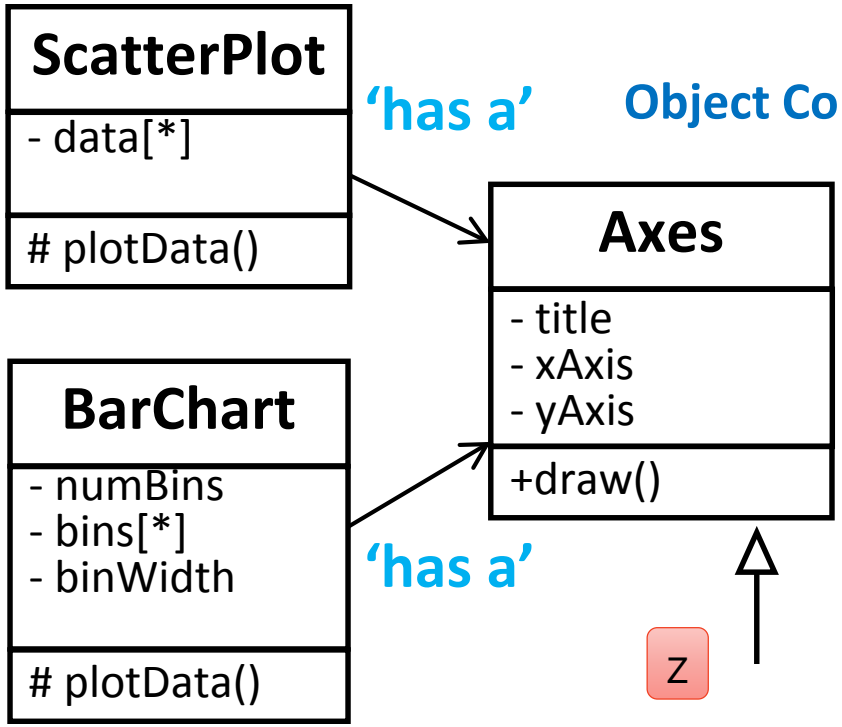
Inheritance vs. Delegation

Refer to video at: 17:45









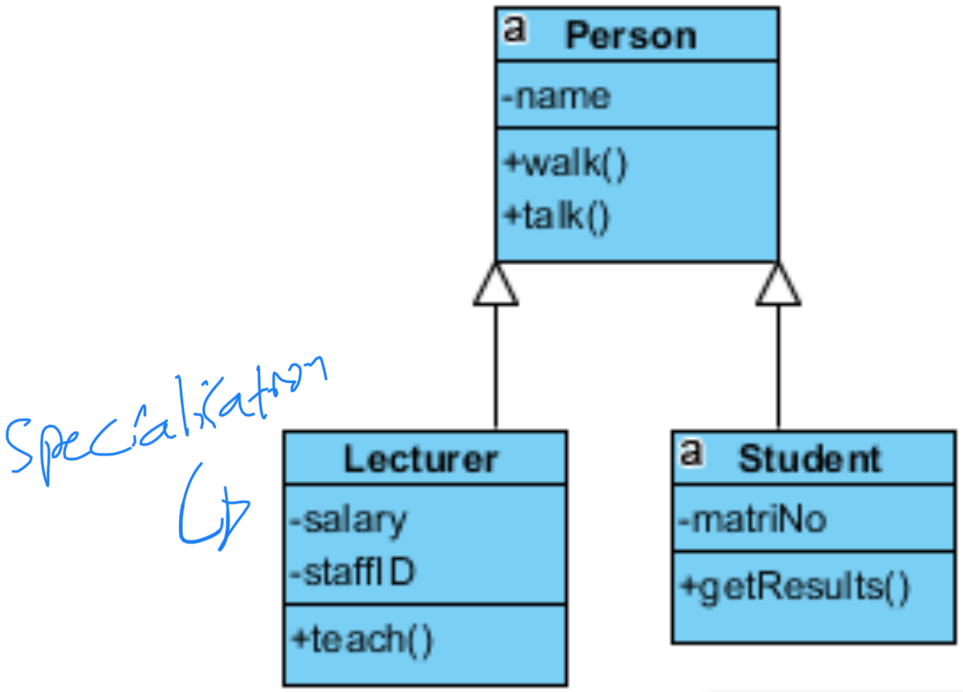
Inheritance vs. Delegation

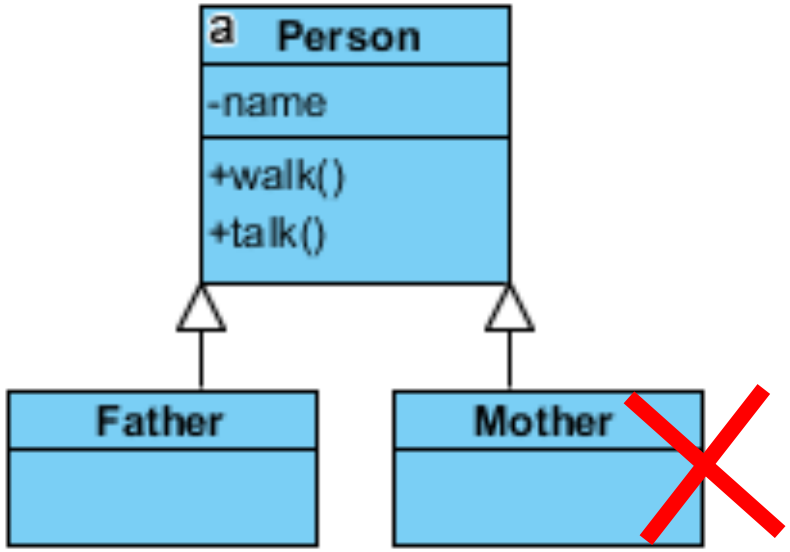
```
/** inheritance
  *   when Graph changes, BarChart needs to be recompiled
  */
class BarChart extends Graph {
    ….
}
```

```
/** delegation
  *   when Axes changes, BarChart does not need to be recompiled
  */
class BarChart {
      private Axes myAxes;
      …
}
```

Inheritance vs. Role

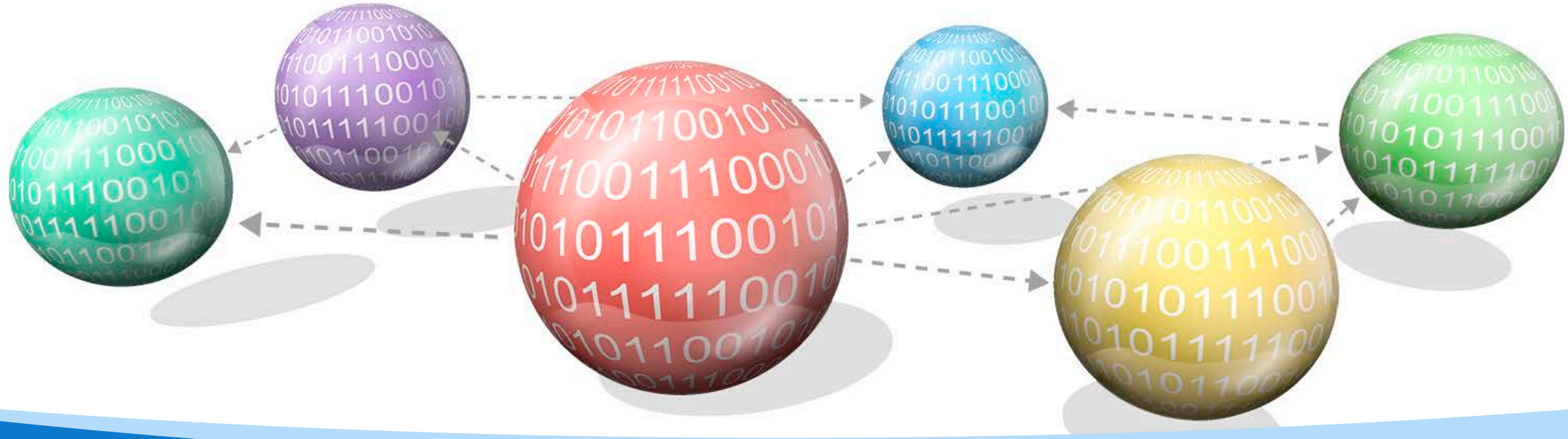
Refer to video at: 21:32






```
Person  father = new Person() ;
Person mother = new Person();
```

```
public  enum FAMILY_ROLE { GRANDFATHER, GRANDMOTHER, FATHER,
MOTHER, FATHER_IN_LAW, MOTHER_IN_LAW, BROTHER, SISTER} ;
```



CE/CZ2002 Object-Oriented Design & Programming

Topic 3: SOLID Design Principles

Chapter 9: Design Principles

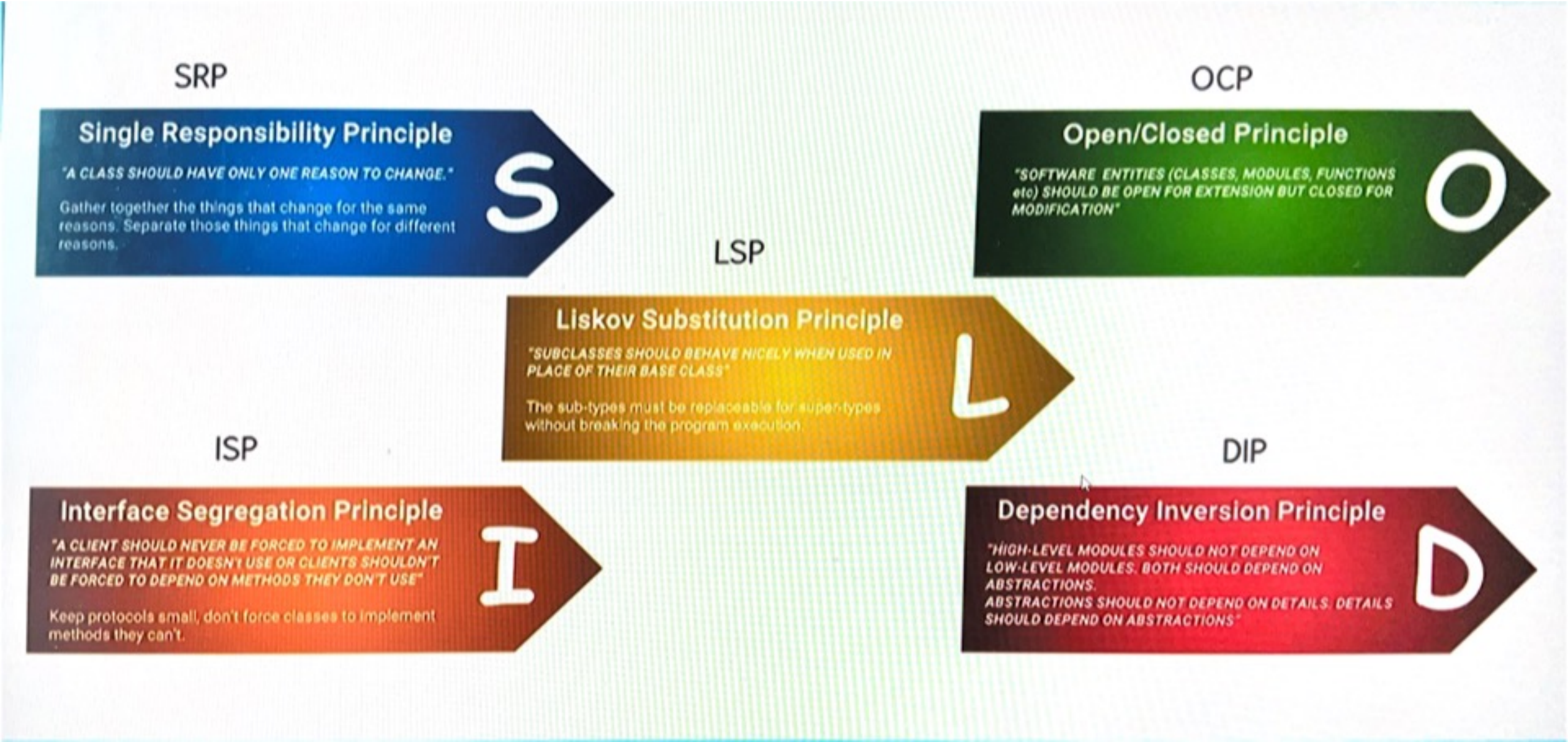
Mr Tan Kheng Leong
Lecturer, School of Computer Science and Engineering

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SOLID Design Principles

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Injection Principle (DIP)

we only focus 5


SRP
Single Responsibility Principle
"A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE."
Gather together the things that change for the same reasons. Separate those things that change for different reasons.
S
OCP
Open/Closed Principle
"SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS etc) SHOULD BE OPEN FOR EXTENSION BUT CLOSED FOR MODIFICATION"
O
LSP
Liskov Substitution Principle
"SUBCLASSES SHOULD BEHAVE NICELY WHEN USED IN PLACE OF THEIR BASE CLASS"
The sub-types must be replaceable for super-types without breaking the program execution.
L
ISP
Interface Segregation Principle
"A CLIENT SHOULD NEVER BE FORCED TO IMPLEMENT AN INTERFACE THAT IT DOESN'T USE OR CLIENTS SHOULDN'T BE FORCED TO DEPEND ON METHODS THEY DON'T USE"
Keep protocols small, don't force classes to implement methods they can't.
I
DIP
Dependency Inversion Principle
"HIGH-LEVEL MODULES SHOULD NOT DEPEND ON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON ABSTRACTIONS.
ABSTRACTIONS SHOULD NOT DEPEND ON DETAILS. DETAILS SHOULD DEPEND ON ABSTRACTIONS"
D

Single Responsibility Principle (SRP)

"There should never be more than ONE reason for a class to change".

- = cohesion
- Each responsibility is an axis of change
- If a class assumes more than one responsibility, then there will be more than one reason for it to change
- If a class has more then one responsibility, then the responsibilities become coupled
- Avoid creating a 'God' class

```java
class Book {
    String name;
    String authorName;
    int year;
    int price;
    String isbn;

    public Book(String name, String authorName, int year, int price, String isbn)
        this.name = name;
        this.authorName = authorName;
        this.year = year;
        this.price = price;
        this.isbn = isbn;
    }
}
```

```java
public class Invoice {

    private Book book;
    private int quantity;
    private double discountRate;
    private double taxRate;
    private double total;

    public Invoice(Book book, int quantity, double discountRate, double taxRate) {
        this.book = book;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.total = this.calculateTotal();
    }

    public double calculateTotal() {
            double price = ((book.price - book.price * discountRate) * this.quantity);

        double priceWithTaxes = price * (1 + taxRate);

        return priceWithTaxes;
    }

    public void printInvoice() {
            System.out.println(quantity + "x " + book.name + " " +          book.price + "$");
            System.out.println("Discount Rate: " + discountRate);
            System.out.println("Tax Rate: " + taxRate);
            System.out.println("Total: " + total);
    }

        public void saveToFile(String filename) {
    // Creates a file with given name and writes the invoice
    }

}
```

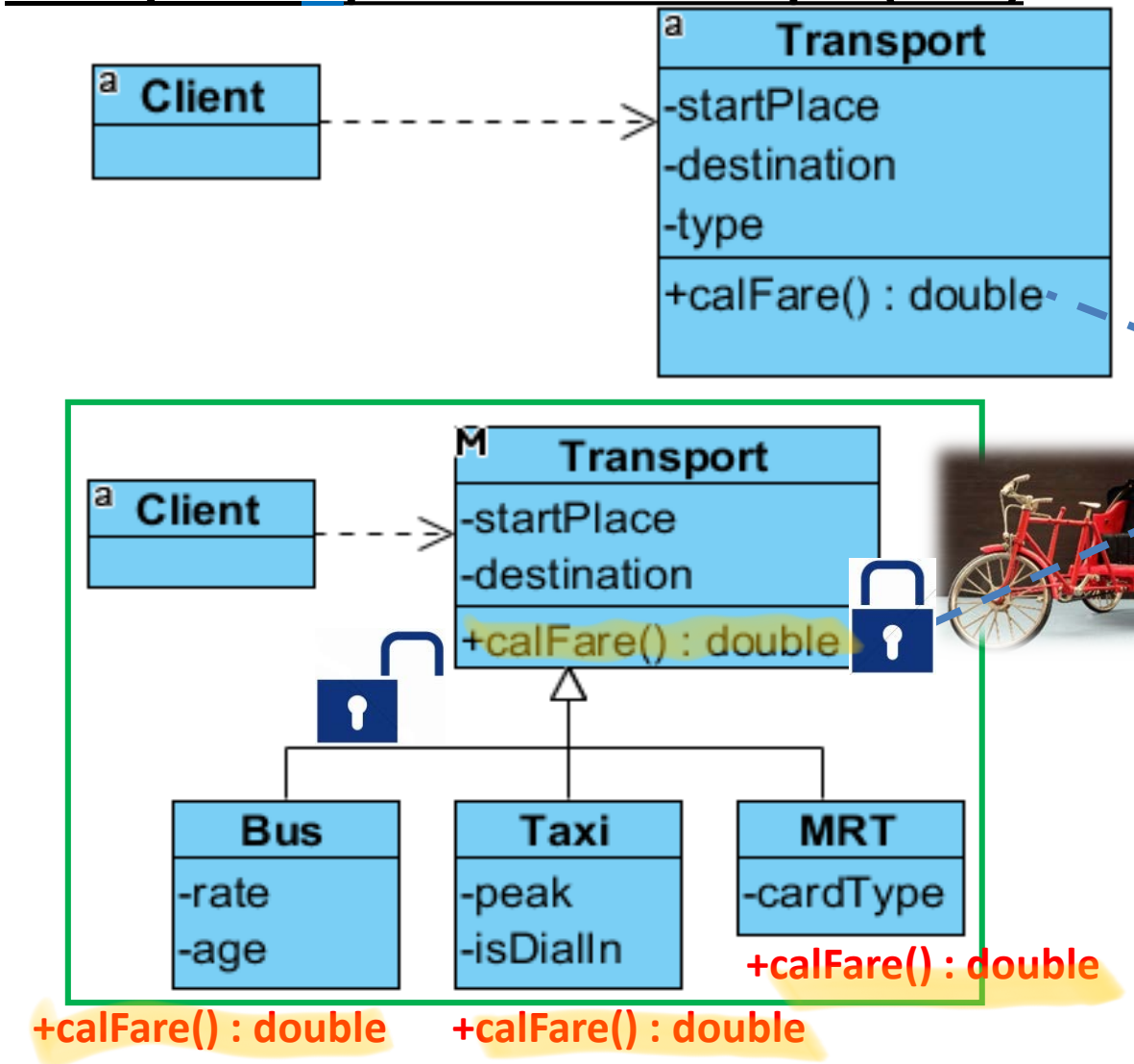
Open-Closed Principle (OCP)

"A module should be open for extension but closed for modification".

- Originated from the work of Bertrand Meyer.
- In other words, we want to be able to change what the modules do, without changing the source code of the modules.
- ***Abstraction is the key to the OCP.***

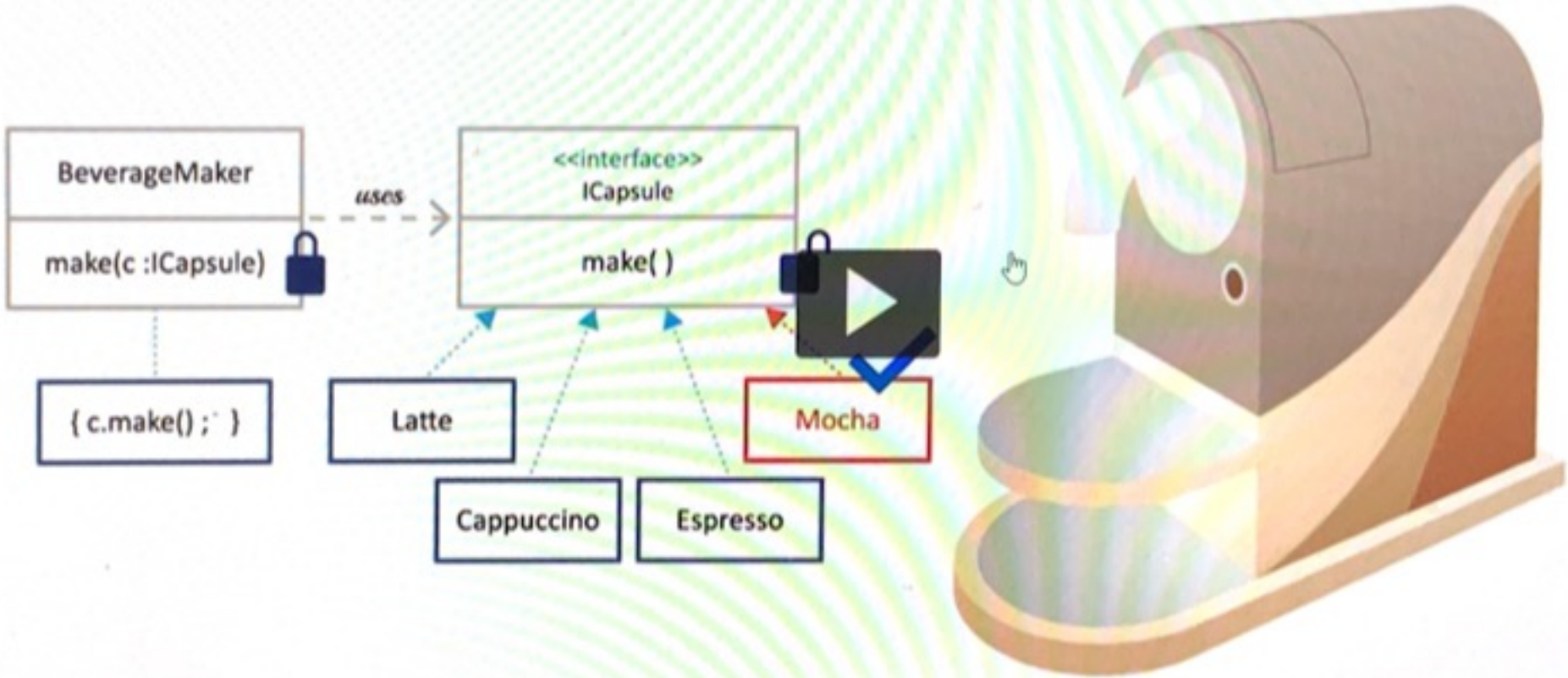
extend but do not change

Example of Open-Closed Principle (OCP)




```
.......
switch (type) {
      case BUS :  .....
      case TAXI : .....
      case MRT : .....
      default :
            // default impl.
            // .......
                  break ;
     return fare;
}
```


Tricycle. Retrieved December 16, 2016 from http://maxpixel.freegreatpicture.com/Rickshaw-Bike-Miniature-Tricycle-1548303.


BeverageMaker
make(c :ICapsule)
uses
<<interface>>
ICapsule
make( )
{ c.make() ; }
Latte
Mocha
Cappuccino
Espresso

Example to explain Open & Closed Principle (OCP)

Download the PDF and click the placeholder to watch an example on OCP.

Liskov Substitution Principle (LSP)

"if for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."

Class	Class Type	Object	substitution
T	superclass	o_2	o_1
S	subclass	o_1	

- If class A is a subtype of class B, we should be able to replace B with A without disrupting the behavior of our program.
- The child class should not implement code such that if it is replaced by the parent class then the application will stop running.

- This principle states that Subclasses or derived classes should be completely substituted of superclass. The Subclass should enhance the functionality but not reduce it.
- Functions that use pointers and reference of base classes must be able to use objects derived classes without knowing it.

superclass ⟵ subclass

- Animal class object we must be able to replace it with the Dog or Cat without exploding our code.

```
public class Animal {
    public void makeNoise()
    {
        System.out.print("Some sound");
    }
}
class Dog extends Animal{
    public void makeNoise()
    {
        System.out.print("Bark");
    }
}

class Cat extends Animal{
    public void makeNoise()
    {
        System.out.print("Meawoo");
    }
}
```

```
class DumbDog extends Animal {

public void makeNoise() {
        throw new RuntimeException("I can't make
noise");
}
}
```

Liskov Substitution Principle (LSP)

"if for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."

- Barbara Liskov first wrote in 1988.
- Subtypes must be substitutable for their base types.
- A **user** of a base class should continue to function properly if a derivative of that base class is passed to it.

User - - - > Sub1

Liskov Substitution Principle (LSP)

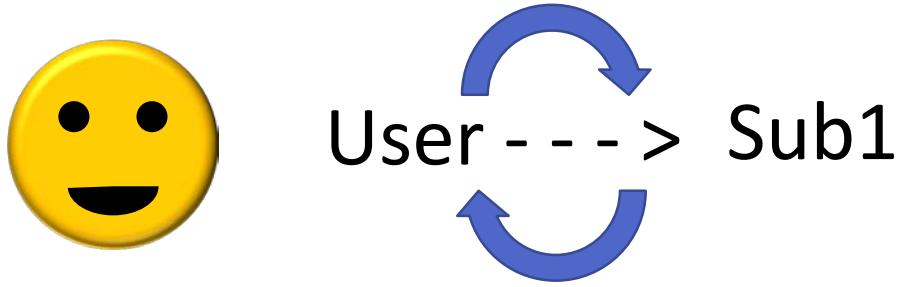
"if for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."

- Barbara Liskov first wrote in 1988.
- Subtypes must be substitutable for their base types.
- A **user** of a base class should continue to function properly if a derivative of that base class is passed to it.

User - - - > Sub2

Liskov Substitution Principle

- **Design by Contract.**
  - Methods (public interfaces) should specify their **pre and post conditions:** what must be true before, and what must be true after their execution, respectively.
- Restating the LSP again, in terms of contracts, a derived class is substitutable for its base class if:
  a) Its pre-conditions are no stronger than the base class method.
  (expect no more)
  a) Its post-conditions are no weaker than the base class method.
  (provide no less)



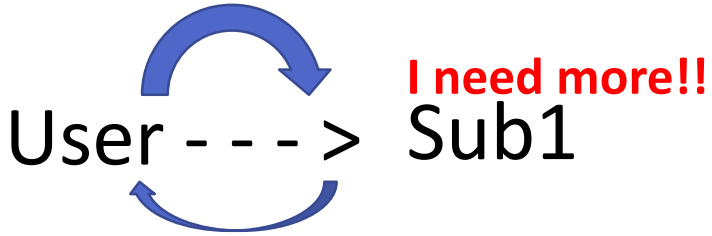
Liskov Substitution Principle

- **Design by Contract.**
  - Methods (public interfaces) should specify their **pre and post conditions:** what must be true before, and what must be true after their execution, respectively.
- Restating the LSP again, in terms of contracts, a derived class is substitutable for its base class if:

a) Its pre-conditions are no stronger than the base class method.
(expect no more)

a) Its post-conditions are no weaker than the base class method.
(provide no less)

User ---> Base

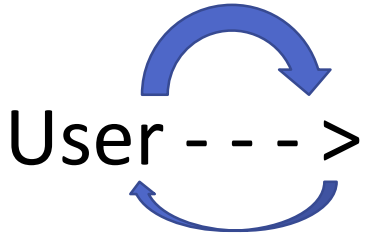


Liskov Substitution Principle

- **Design by Contract.**
  - Methods (public interfaces) should specify their **pre and post conditions:** what must be true before, and what must be true after their execution, respectively.
- Restating the LSP again, in terms of contracts, a derived class is substitutable for its base class if:

a) Its pre-conditions are no stronger than the base class method.
(expect no more)

a) Its post-conditions are no weaker than the base class method.
(provide no less)



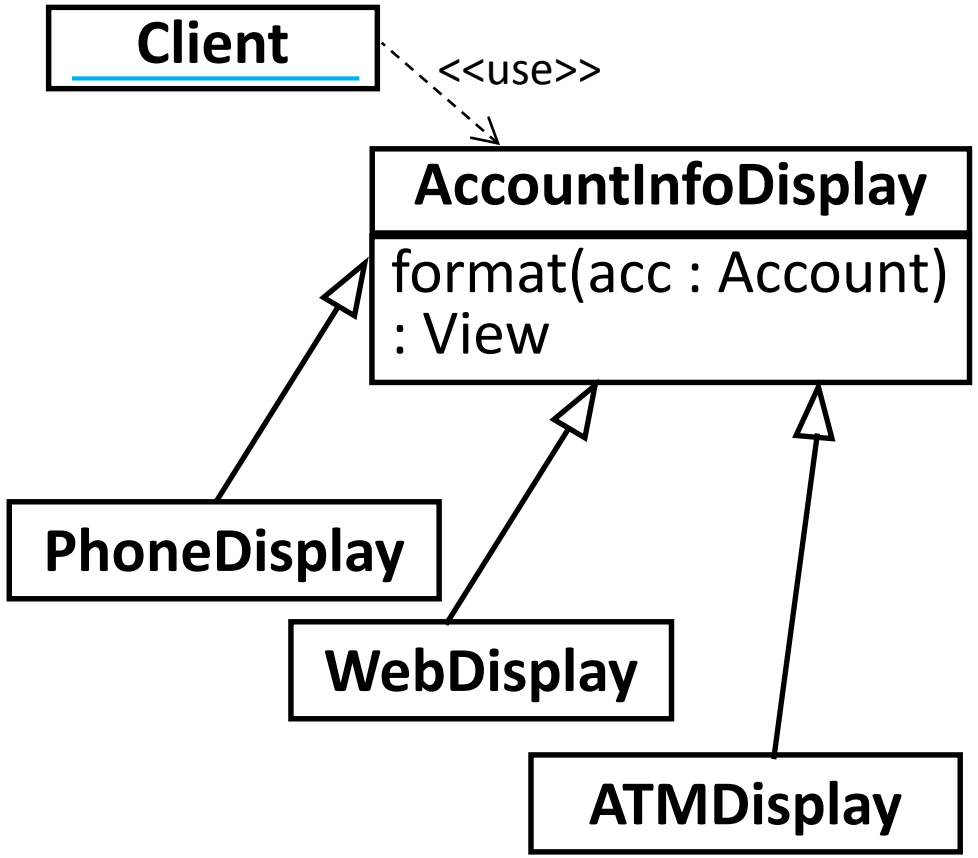
I need more!!

~~$3~~

$4

Noodle soup. Retrieved December 16, 2016 from https://upload.wikimedia.org/wikipedia/commons/a/af/Fish_ball_soup_noodle.jpg.

Example of Liskov Substitution Principle




```
public class AccountInfoDisplay {
   .......
   public View  format(Account acc) {
        if (acc == null)  return ;
        // code to format and return full
        // details of acc View
   }
}
```

```
public class ATMDisplay extends
AccountInfoDisplay {
   .......
   public View  format(Account acc) {
      // assume client check for acc not
null
        // code to format and return
         //partial details of acc View
   }
}
```

Subtype expecting more
=> Violation of LSP

Subtype providing less
=> Violation of LSP

Example of Liskov Substitution Principle

```
public class AccountInfoDisplay {
   .....
   public View  format(Account acc) {
        if (acc == null)  return ;
        // codes to format and return full
        // details of acc View
   }
}
```

```
public class ATMDisplay extends
AccountInfoDisplay {
   .....
   public View  format(Account acc) {
        // assume client check for acc not
null
        // codes to format and return
         //partial details of acc View
   }
}
```

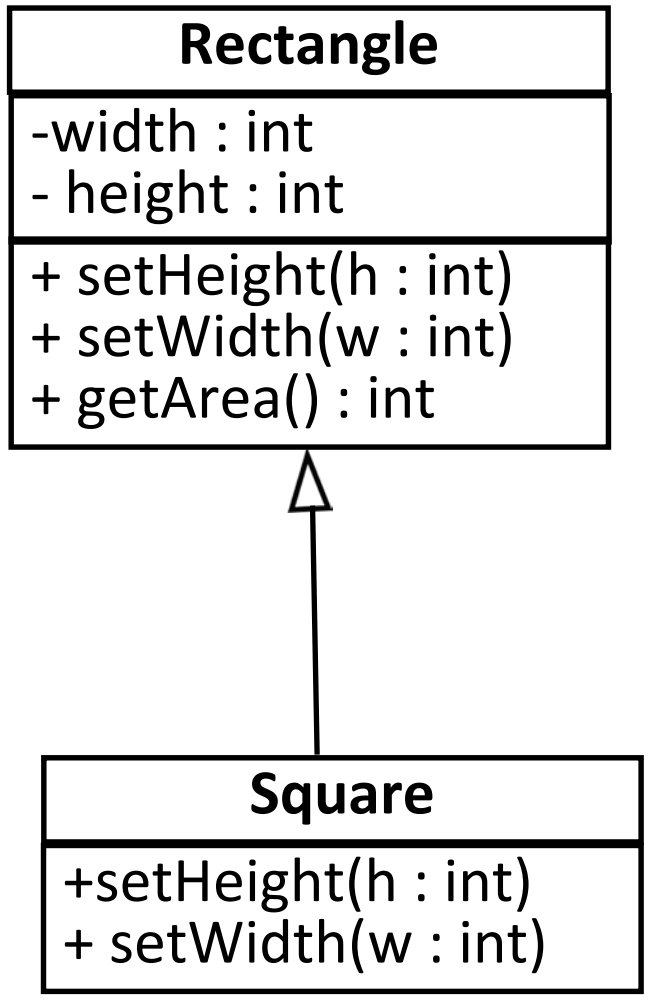
	superClass	subClass
Error checking	if (acc == null) return ;	N/A
Return information	full details of acc View	partial details of acc View

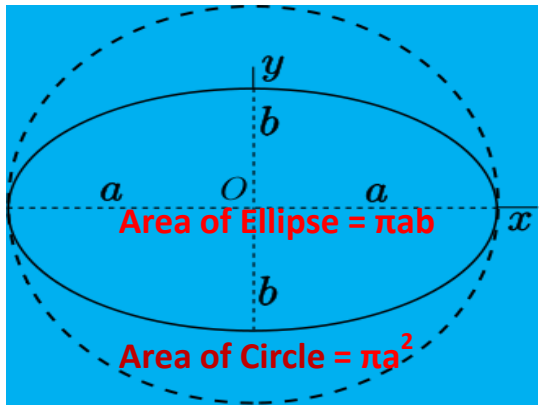
Liskov Substitution Principle (LSP)

Subclass must do all the things super class do

Subclass must not bring any trouble that super class don't

Liskov Substitution Principle: Classic Example






```
public class Square extends Rectangle {
  .......
  public void  setWidth(w : int) {
        width = w ;  height = w ;
  }
  public void  setHeight(h : int) {
        width = h ;  height = h ;
  }
}
```

```
public class Client{
  public static Rectangle getNewRectangle()
     {  return new Square() ;  }
  public static void main (String args[]) {
     Rectangle r = Client.getNewRectangle();
     r.setWidth(5);
     r.setHeight(10); // client thinks that r is a rectangle.
           System.out.println(r.getArea());
  } // output 100 instead of 50
}
```

Liskov Substitution Principle: Classic Example

Verify your sub class

with base class

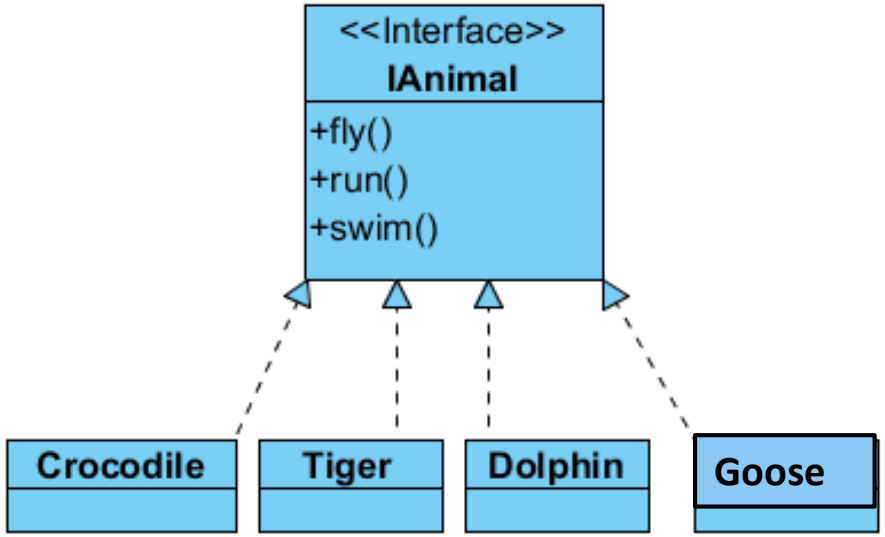
Refer to video at: 19:02

Interface Segregation Principle (ISP)

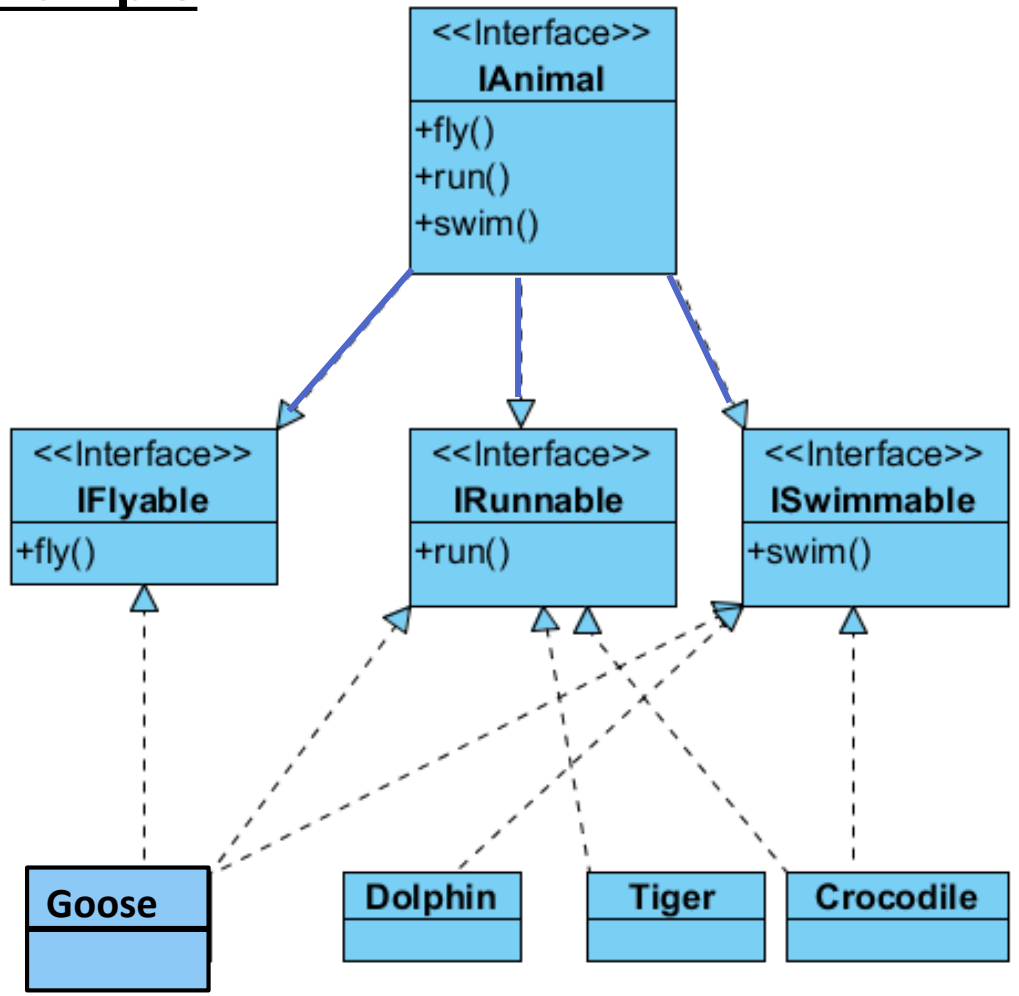
"Many client specific interfaces are better than one general purpose interface".

- ***Classes should not depend on interfaces that they do not use.***
- Avoid FAT Interfaces.

Interface Segregation Principle (ISP) example

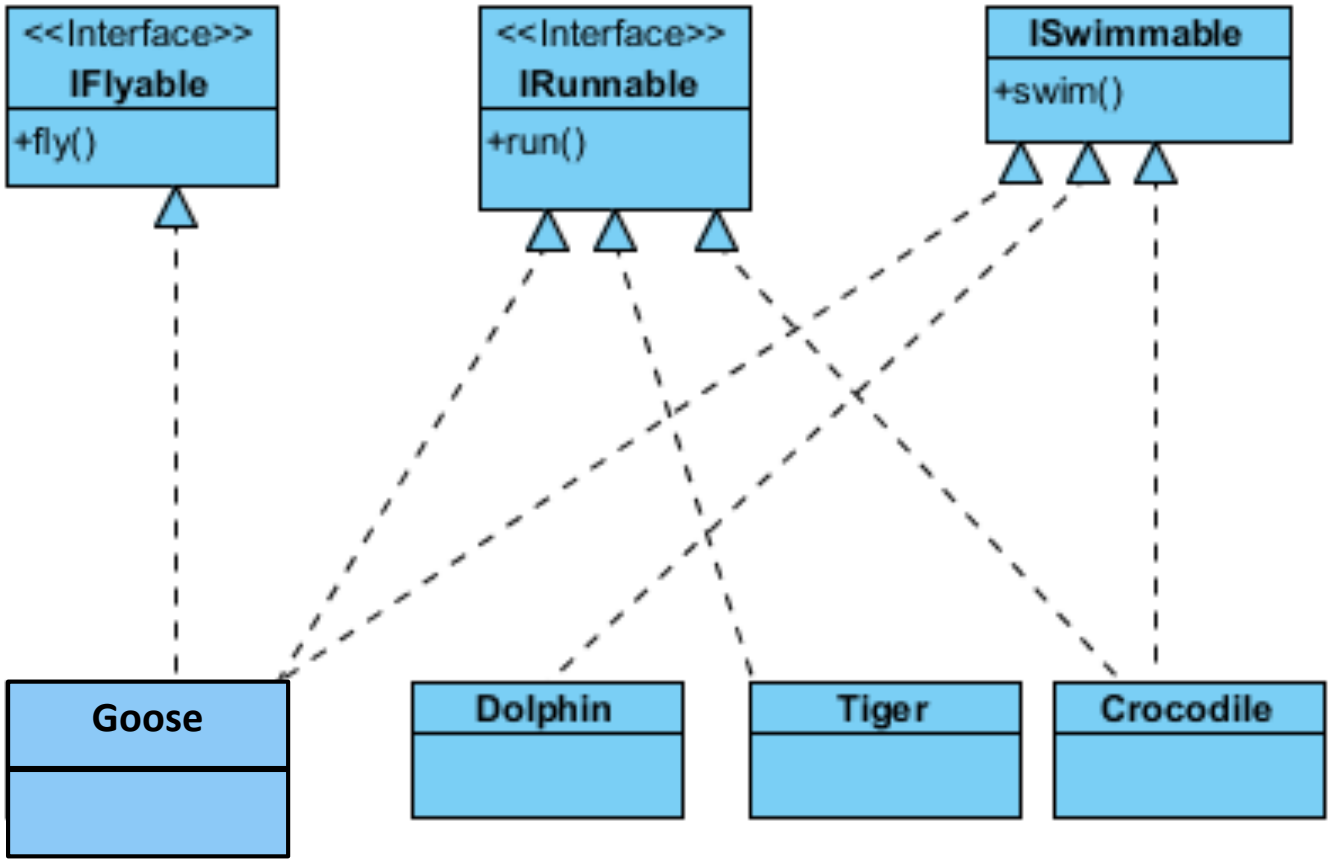


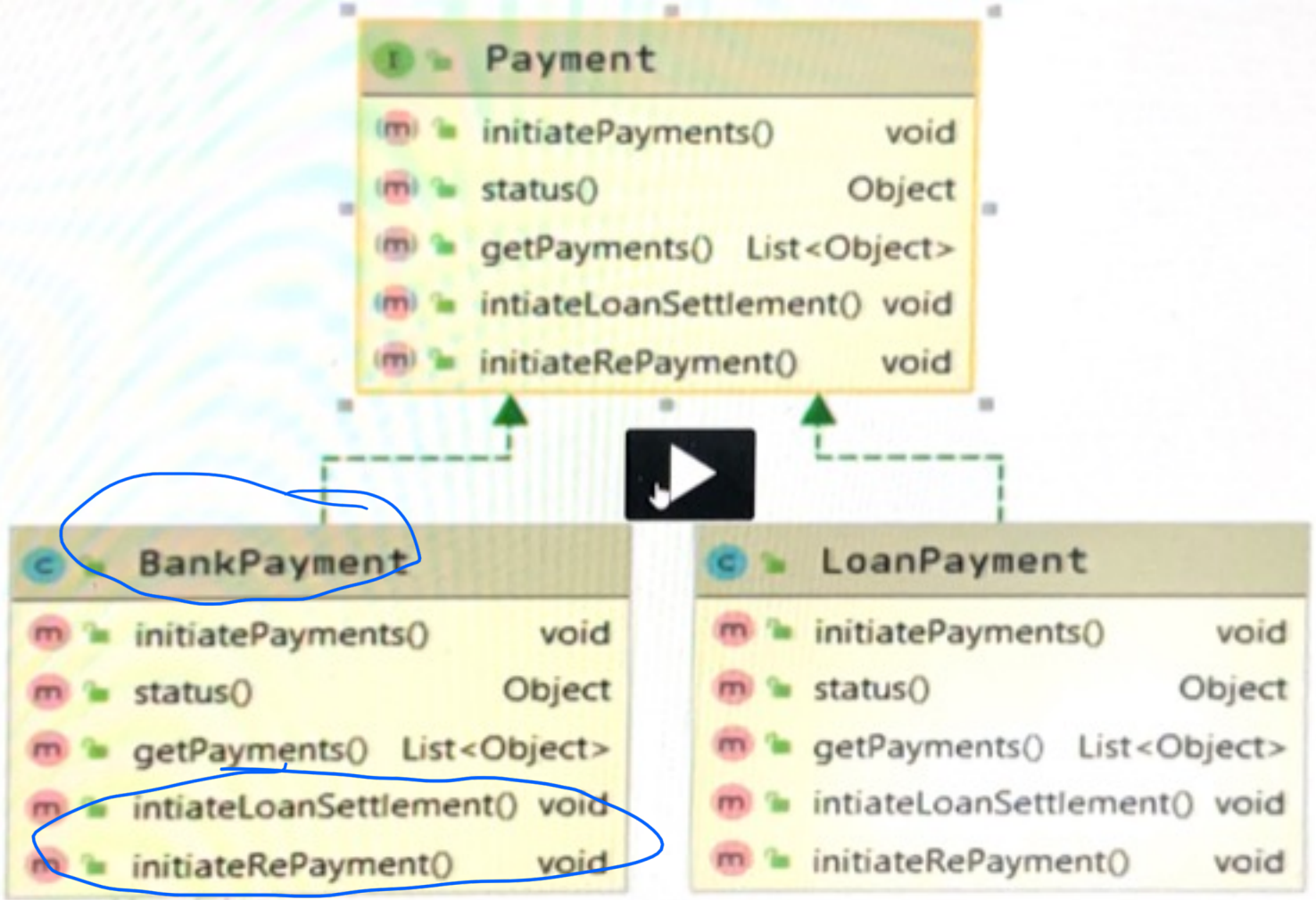

```
public class Tiger  implements IAnimal{
    public void fly { }
    public void run { // code added }
    public void swim { }
}
```



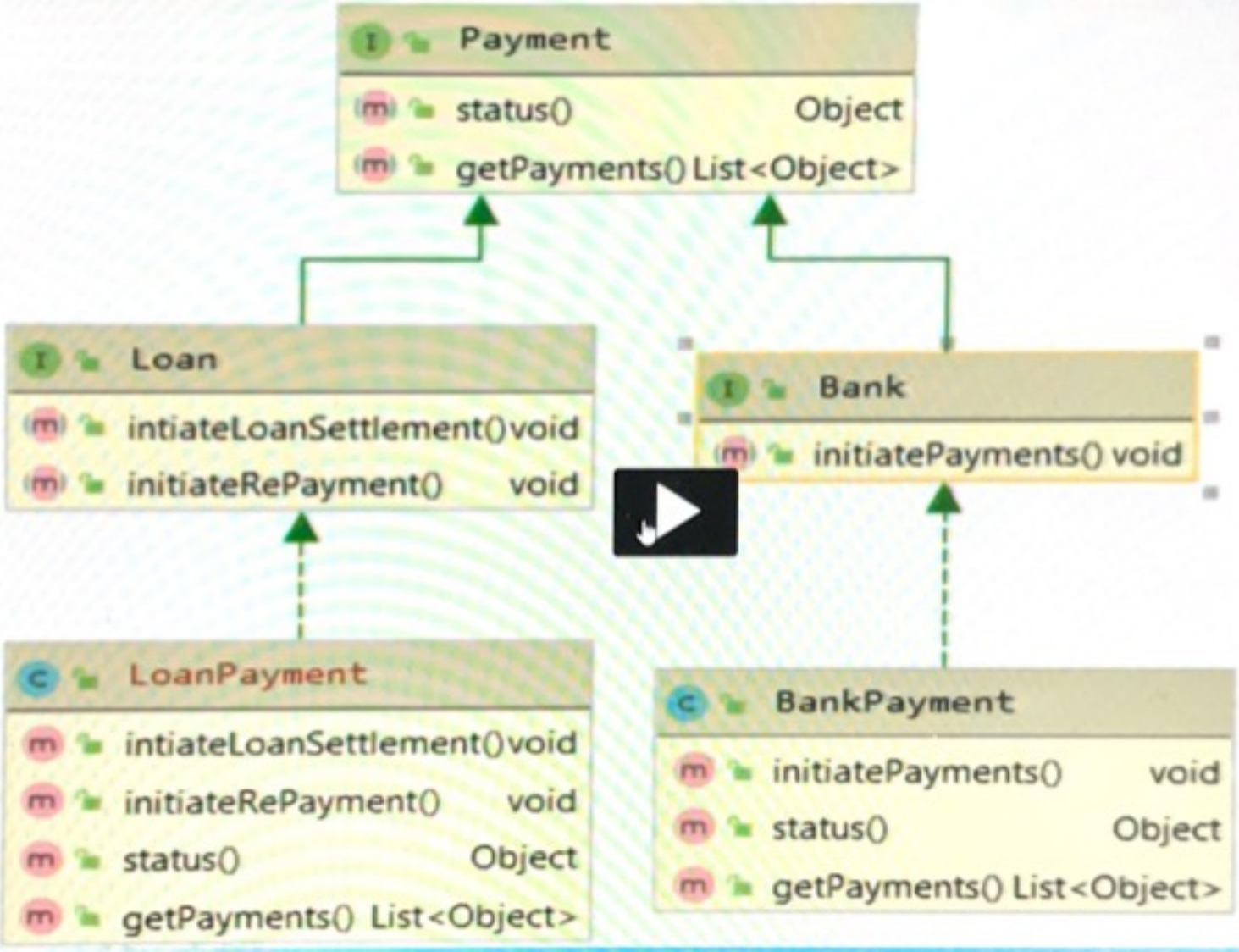
Interface Segregation Principle (ISP)

Seperate when necessary




Payment
initiatePayments() void
status() Object
getPayments() List<Object>
intiateLoanSettlement() void
initiateRePayment() void
BankPayment
initiatePayments() void
status() Object
getPayments() List<Object>
intiateLoanSettlement() void
initiateRePayment() void
LoanPayment
initiatePayments() void
status() Object
getPayments() List<Object>
intiateLoanSettlement() void
initiateRePayment() void

```java
public class BankPayment implements Payment {

    @Override
    public void initiatePayments() {
        // ...
    }

    @Override
    public Object status() {
        // ...
    }

    @Override
    public List<Object> getPayments() {
        // ...
    }

    @Override
    public void intiateLoanSettlement() {
        throw new UnsupportedOperationException("This is not a loan payment");
    }

    @Override
    public void initiateRePayment() {
        throw new UnsupportedOperationException("This is not a loan payment");
    }
```


Payment
status() Object
getPayments() List<Object>
Loan
intiateLoanSettlement() void
initiateRePayment() void
Bank
initiatePayments() void
LoanPayment
intiateLoanSettlement() void
initiateRePayment() void
status() Object
getPayments() List<Object>
BankPayment
initiatePayments() void
status() Object
getPayments() List<Object>

Dependency Injection Principle (DIP)

A. A HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

- When high level modules are independent of the low level modules, the high level modules can be reused quite simply.
- It is the principle at the very heart of Framework design.
- Also known as **Inversion of Control (IoC).**

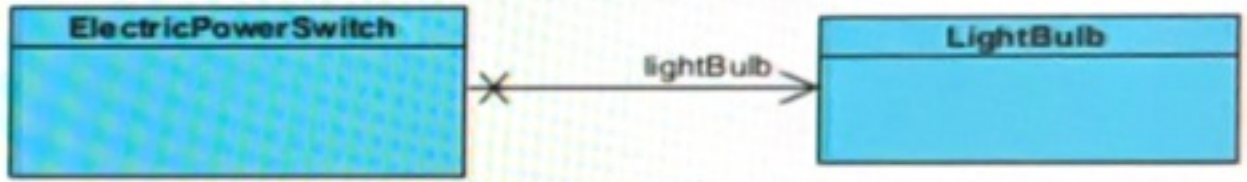
LightBulb.java

```
public class LightBulb {
public void turnOn() {
System.out.println("LightBulb: Bulb turned on...");
}
public void turnOff() {
System.out.println("LightBulb: Bulb turned off...");
}
}
```

ElectricPowerSwitch.java

```
public class ElectricPowerSwitch {
public LightBulb lightBulb;
public boolean on;
public ElectricPowerSwitch(LightBulb lightBulb) {
this.lightBulb = lightBulb;
this.on = false;
}
public boolean isOn() {
return this.on;
}
public void press(){
boolean checkOn = isOn();
if (checkOn) {
lightBulb.turnOff();
this.on = false;
} else {
lightBulb.turnOn();
this.on = true;
}
}
}
```

Problem?



- Our high-level ElectricPowerSwitch class is directly dependent on the low-level LightBulb class.
- The LightBulb class is hardcoded in ElectricPowerSwitch.
  - But, a switch should not be tied to a bulb. It should be able to turn on and off other appliances and devices too, say a fan, an AC, or the entire lightning system of an amusement park.
- Now, imagine the modifications we will require in the ElectricPowerSwitch class each time we add a new appliance or device.
- We can conclude that our design is flawed and we need to revisit it by following the DIP.

```
public interface
Switch {
boolean isOn();
void press();
}
```

An interface for switches with the isOn() and press() methods. This interface will give us the flexibility to plug in other types of switches, say a remote control switch later on

```
public interface
Switchable {
void turnOn();
void turnOff();
}
```

The Switchable interface with the turnOn() and turnoff() methods. From now on, any switchable devices in the application can implement this interface and provide their own functionality.

DIP

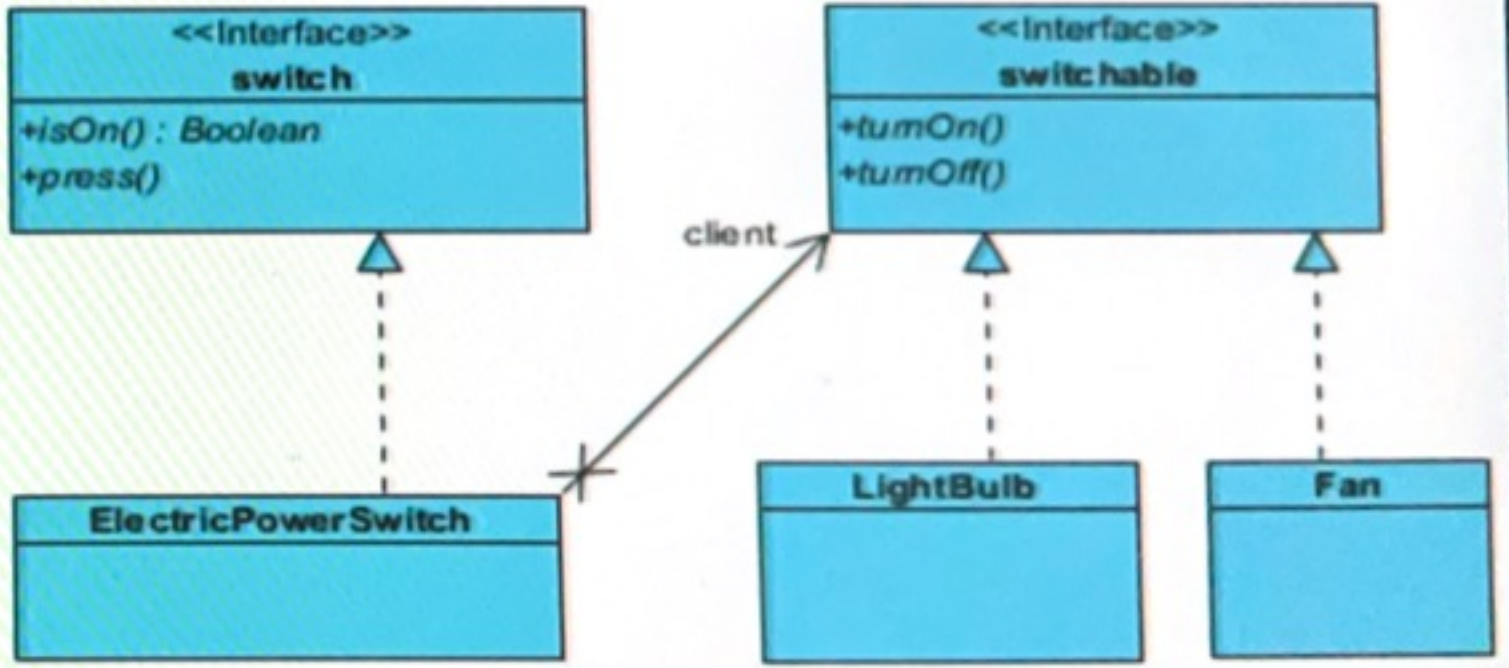
```
public interface Switch {
boolean isOn();
void press();
}
```

```
public interface Switchable {
void turnOn();
void turnOff();
}
```

```
public class ElectricPowerSwitch implements Switch {
public Switchable client;
public boolean on;
public ElectricPowerSwitch(Switchable client) {
this.client = client;
this.on = false;}
public boolean isOn() {
return this.on;}
public void press(){
boolean checkOn = isOn();
if (checkOn) {
client.turnOff();
this.on = false;
} else {
client.turnOn();
this.on = true;
}}}
```

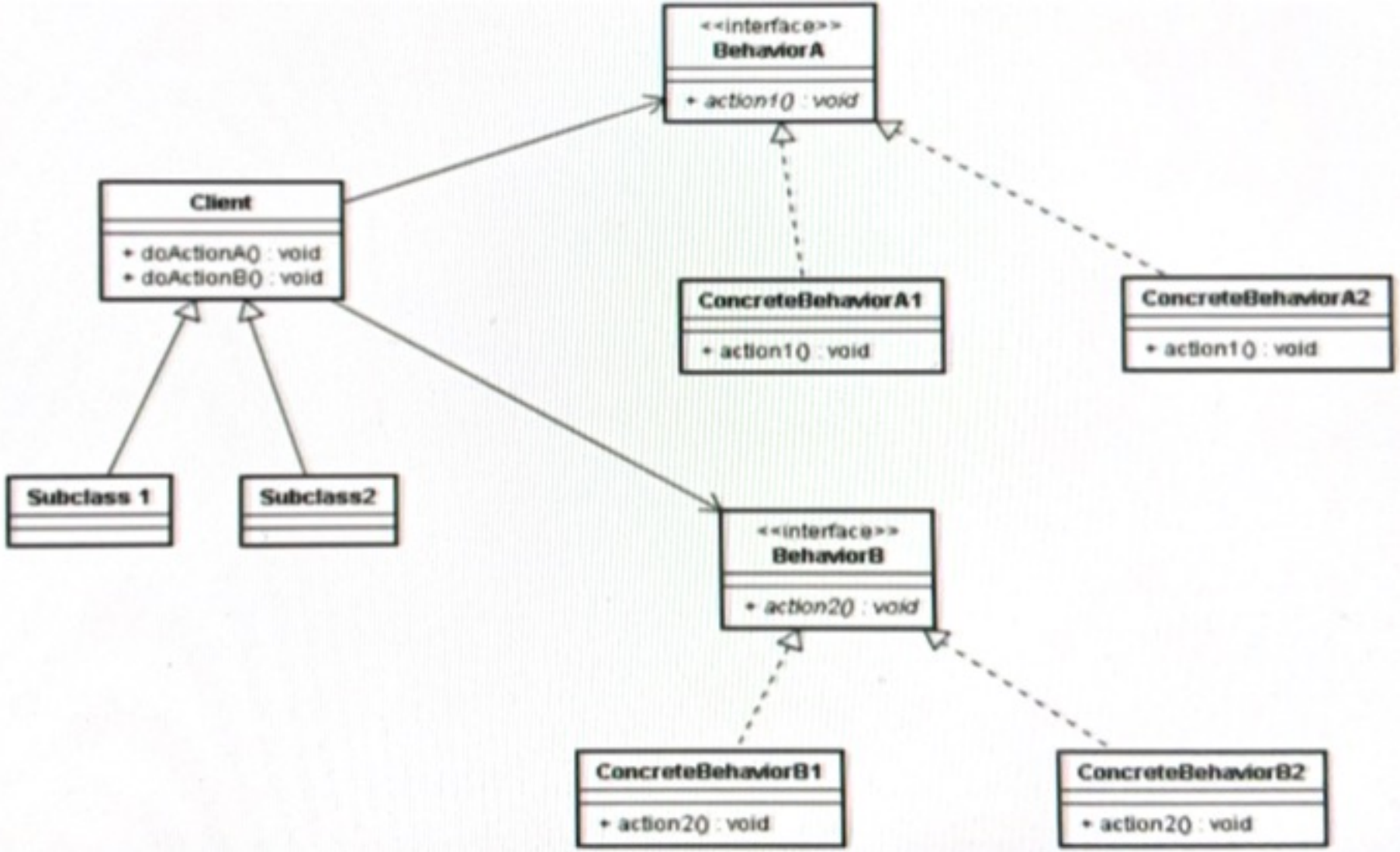
```
public class LightBulb implements Switchable {
@Override
public void turnOn() {
System.out.println("LightBulb: Bulb turned on...");
}
@Override
public void turnOff() {
System.out.println("LightBulb: Bulb turned off...");
}
}
```

```
public class Fan implements Switchable {
@Override
public void turnOn() {
System.out.println("Fan: Fan turned on...");
}
@Override
public void turnOff() {
System.out.println("Fan: Fan turned off...");
}
}
```



Test

```
public class ElectricPowerSwitchTest {

    public void testPress() throws Exception {

    Switchable switchableBulb=new LightBulb();
    Switch bulbPowerSwitch=new ElectricPowerSwitch(switchableBulb);
    bulbPowerSwitch.press();
    bulbPowerSwitch.press();

    Switchable switchableFan=new Fan();
    Switch fanPowerSwitch=new ElectricPowerSwitch(switchableFan);
    fanPowerSwitch.press();
    fanPowerSwitch.press();
    }
}
```

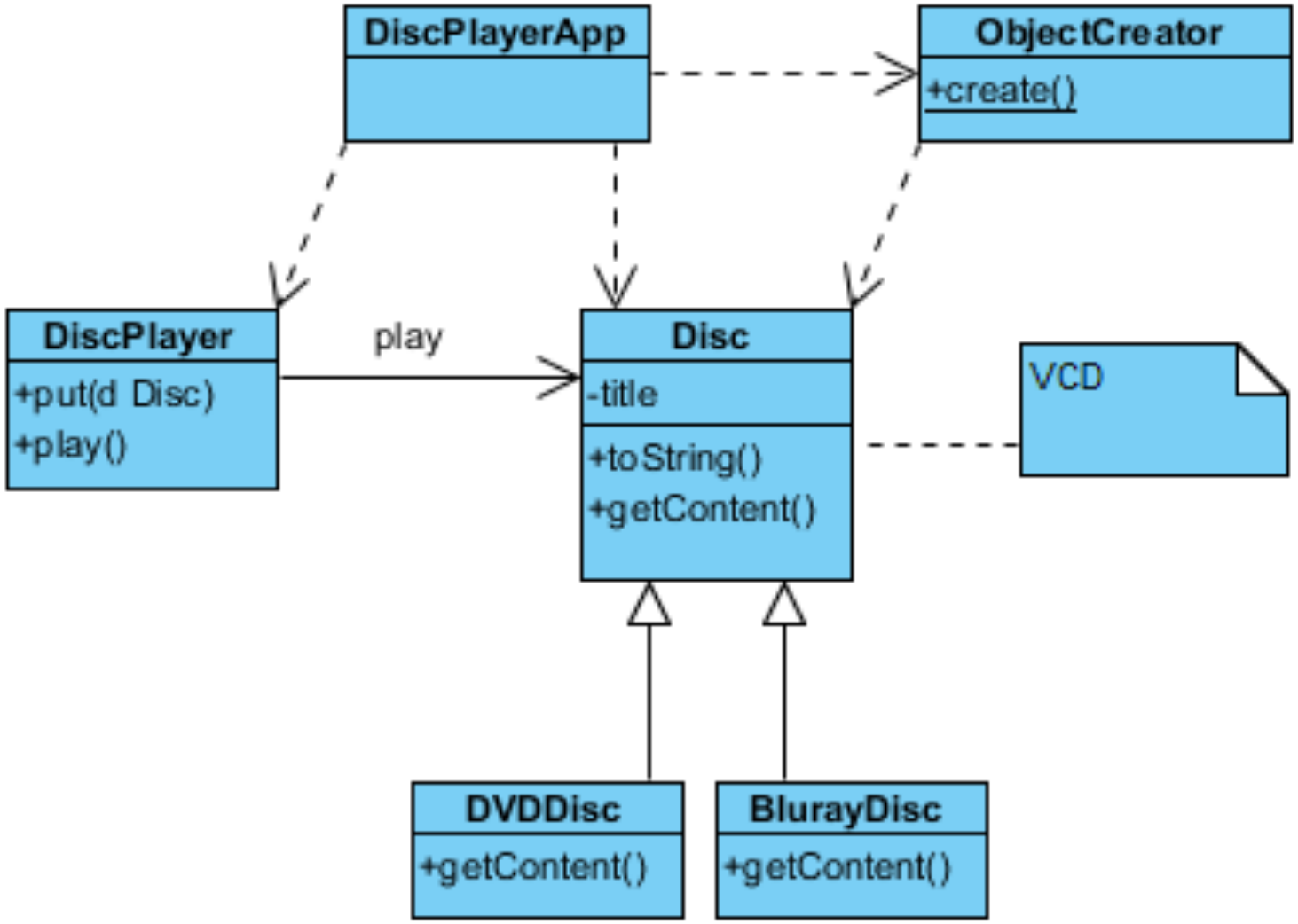

«interface»
BehaviorA
+ action10 : void
Client
+ doActionA0 : void
+ doActionB0 : void
ConcreteBehaviorA1
+ action10 : void
ConcreteBehaviorA2
+ action10 : void
Subclass 1
Subclass2
«interface»
BehaviorB
+ action20 : void
ConcreteBehaviorB1
+ action20 : void
ConcreteBehaviorB2
+ action20 : void

Dependency Injection Principle (DIP) example

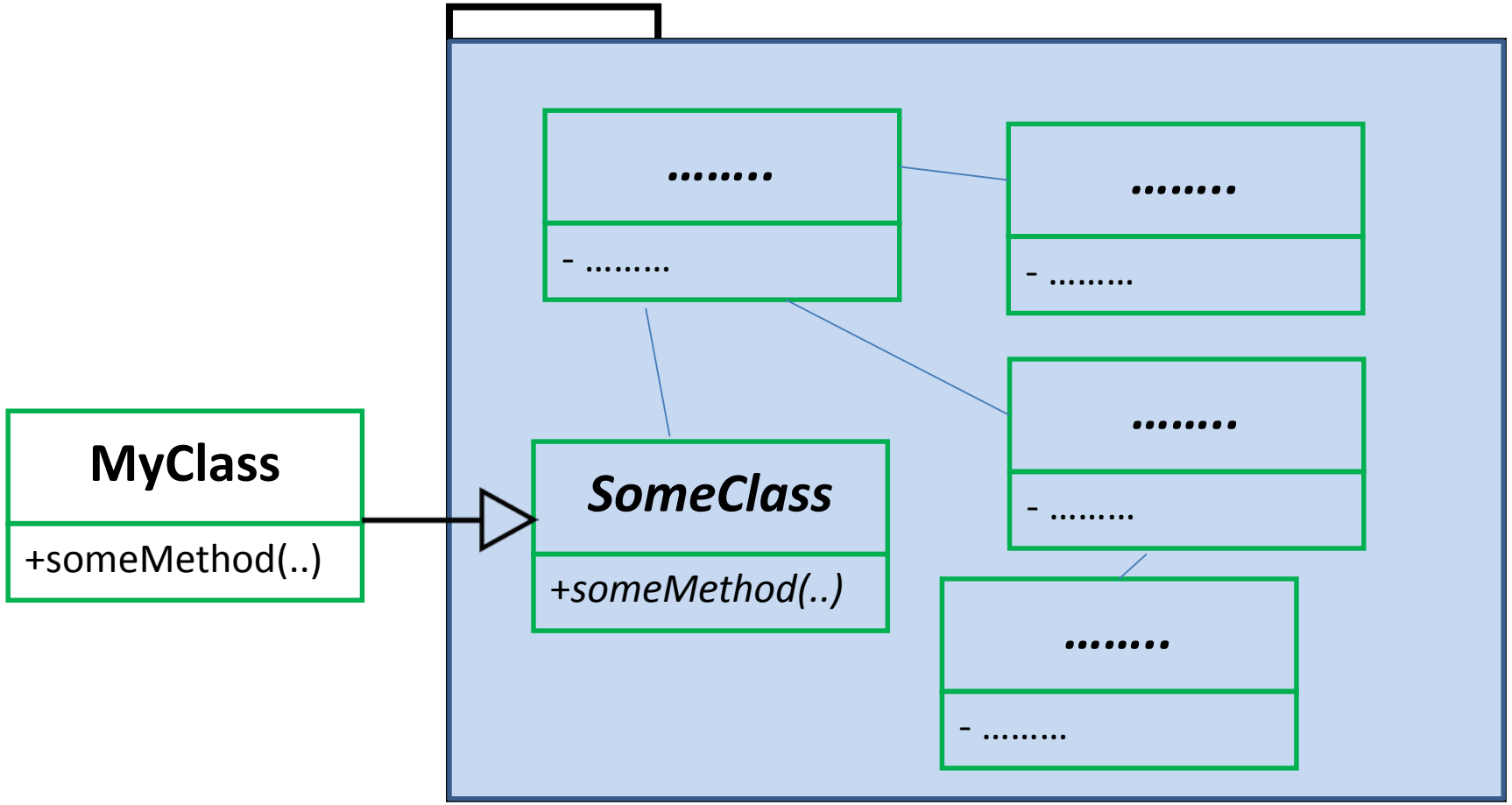
- Using Reflection classes
- Mini Framework
- Plugging into Framework via Interface or Abstract class

Demo

Refer to video at: 27:03



Some Framework Package of Classes



Frameworks are large bodies (usually many classes) of prewritten code to which you add your own class to extend/realise framework's class/interface to solve a problem in a specific domain. The framework that is in control, instantiates your class. You make use of a framework by calling its methods and supplying "callbacks".

Design Patterns

- Gang of Four
  - The four authors were Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- Proven Solution to known problem
- Examples
  - Singleton
  - Adapter
  - Observer
  - ……and more.. http://en.wikipedia.org/wiki/Software_design_pattern
  - Model –View-Controller (MVC)
    - Widely used pattern for the web

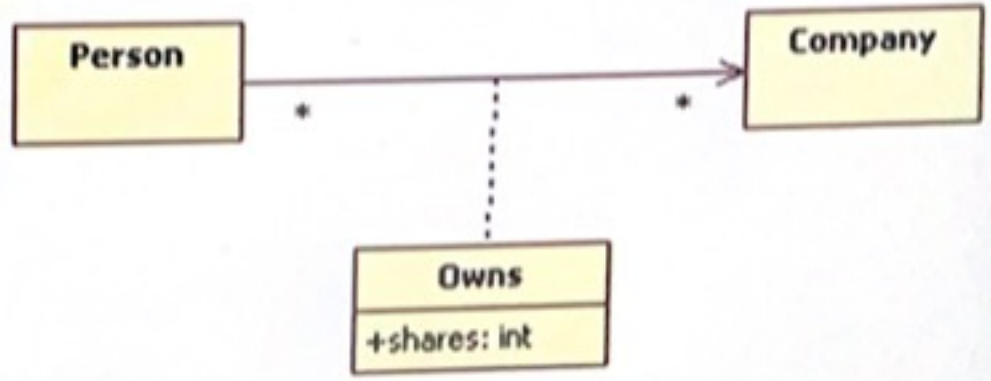
Further Reading

Principles and Patterns

- Principles_and_Patterns.pdf in NTULearn

During the implementation phase an association class might be ...
into Java as follows:

```
public class Company { ... }

public class Person {
   private List<Owns> invstments;
   public void add(Company c, int shares) {
      investments.add(new Owns(c, this, shares);
   }
   // etc.
}

class Owns {
   private Person owner;
   private Company company;
   private int shares;
   public Owns(Company c, Person p, int num) {
      comnpany = c;
      person = p;
      shares = num;
   }
   // etc.
}
```



http://www.cs.sjsu.edu/~pearce/modules/lectures/uml/class/AssociationClasse