Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 2
Diseño de Componentes

IIC2113 – Diseño Detallado de Software

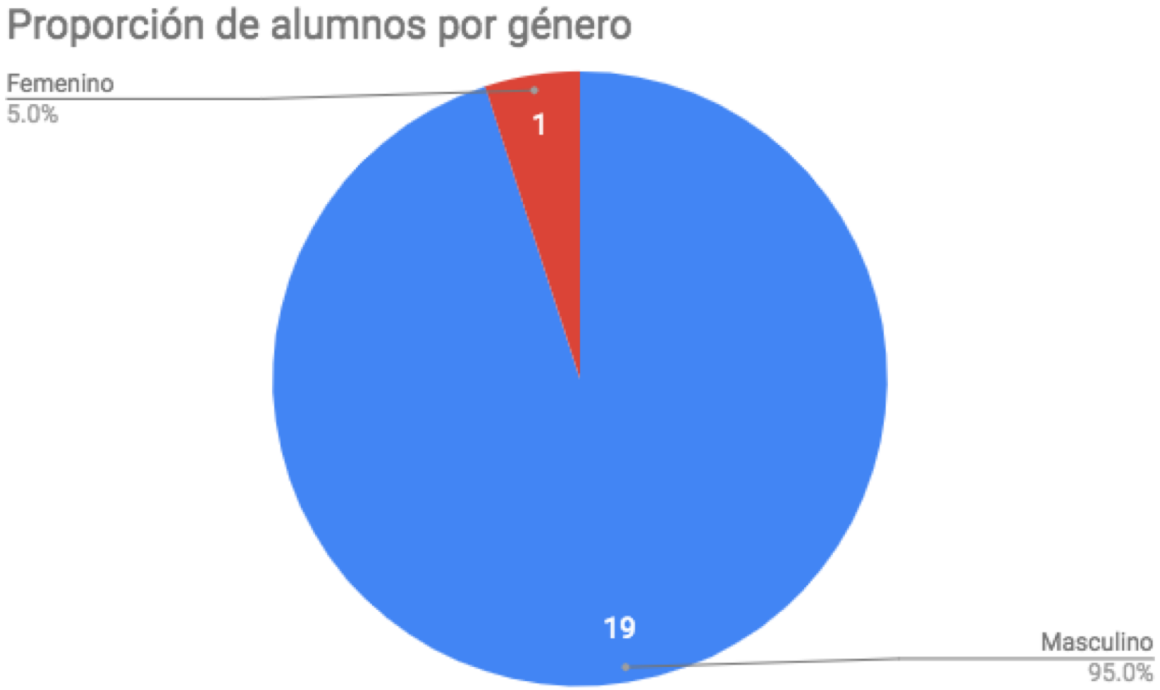
Rodrigo Saffie

rasaffie@uc.cl

24 de agosto de 2018

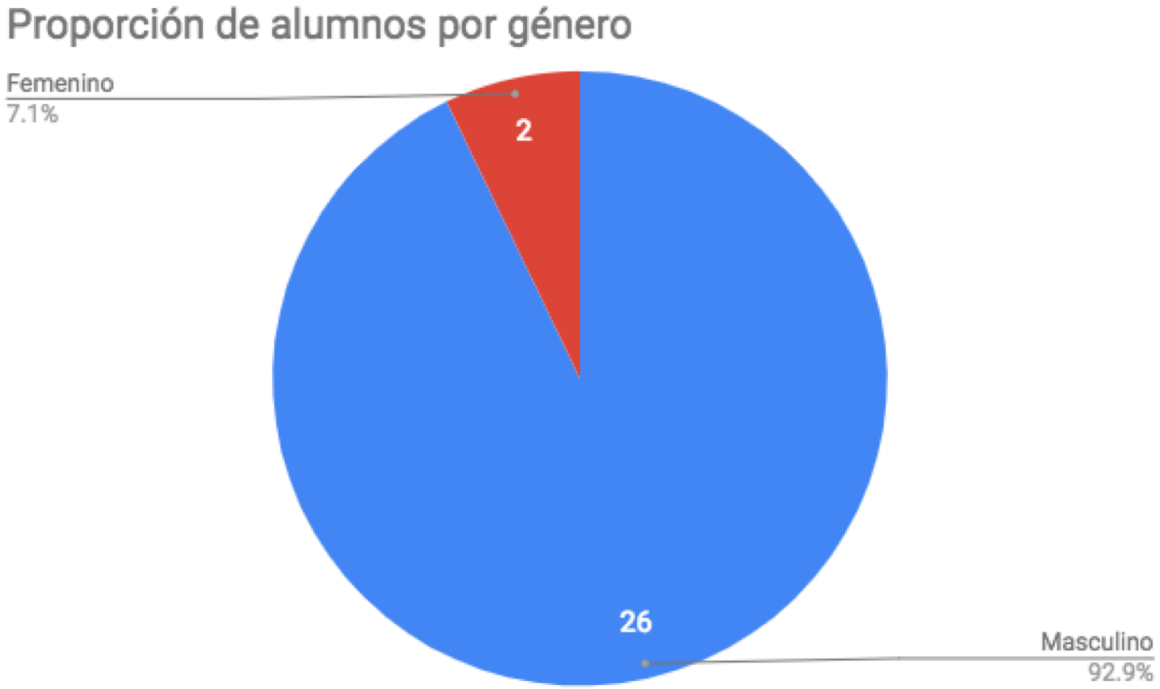
Estadísticas del curso

- 2016 – 2: 20 alumnos



Estadísticas del curso

- 2017 – 2: 28 alumnos



Estadísticas del curso

- 2018 – 2: 70 alumnos

Estadísticas del curso

- GitHub

https://www.research.net/r/97ZLPMH

Actividad 1

Modelo 4 + 1:

- Diagrama de clases
- Diagrama de secuencia
- Diagrama de actividad
- Diagrama de componentes
- Diagrama de despliegue
- Diagrama de casos de uso

Diseño de componentes de software

Conceptos básicos:

- Herencia: Mecanismo a través del cual una clase **Y** hereda los atributos y métodos de una clase **X**.

```ruby
class Bird
  def preen
    puts "I am cleaning my feathers."
  end
  def fly
    puts "I am flying."
  end
end

class Penguin < Bird
  def fly
    puts "Sorry. I'd rather swim."
  end
end
```

Diseño de componentes de software

Conceptos básicos:

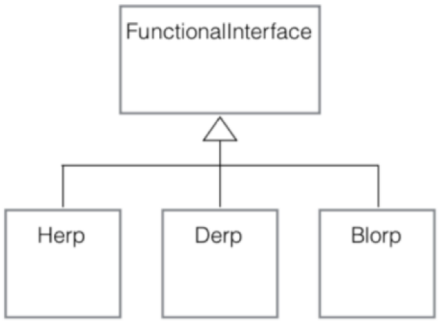
- Herencia: Mecanismo a través del cual una clase **Y** hereda los atributos y métodos de una clase **X**.
- Promueve la reusabilidad, disminuyendo costos de desarrollo y mantenibilidad.
- Su mal uso puede conllevar a todo lo contrario.
  - Dificulta legibilidad
  - Introduce acoplamiento entre padres e hijos.

Diseño de componentes de software

Conceptos básicos:

- Interfaz: Colección de operaciones (métodos) que las clases concretas deben implementar.
- Se pueden interpretar como un contrato.

```
class FunctionalInterface
  def do_thing
    raise "This is not implemented!"
  end
end

class Herp < FunctionalInterface
  def do_thing
    puts "herp"
  end
end

class Derp < FunctionalInterface
  def do_thing
    puts "derp"
  end
end

class Blorp < FunctionalInterface
  def do_some_other_thing
    puts "whoops"
  end
end
```



Diseño de componentes de software

Conceptos básicos:

- Interfaz: Colección de operaciones (métodos) que las clases concretas deben implementar.
- Se pueden interpretar como un contrato.
- Facilitan la extensibilidad del código

¿Cómo diseñar componentes de *software*?

Principios *S.O.L.I.D.*

- Propuestos por Robert C. Martin (cerca del 2000)
- Fomentan *software* fácil de mantener y extender

¿Cómo diseñar componentes de *software*?

Principios *S.O.L.I.D.*

- **S** – *Single-responsibility*
- **O** – *Open-Closed*
- **L** – *Liskov substitution*
- **I** – *Interface segregation*
- **D** – *Dependency Inversion*

Single-Responsibility principle

"A class should have only a single responsibility (one reason to change)"

Single-Responsibility principle

"A class should have only a single responsibility (one reason to change)"

```
public class InvitationService
{
    public void SendInvite(string email, string firstName, string lastName)
    {
        if(String.IsNullOrWhiteSpace(firstName) || String.IsNullOrWhiteSpace(lastName))
        {
            throw new Exception("Name is not valid!");
        }

        if(!email.Contains("@") || !email.Contains("."))
        {
            throw new Exception("Email is not valid!!");
        }
        SmtpClient client = new SmtpClient();
        client.Send(new MailMessage("mysite@nowhere.com", email) { Subject = "Please join me at
    }
}
```

Single-Responsibility principle

"A class should have only a single responsibility (one reason to change)"

```
public class UserNameService
{
    public void Validate(string firstName, string lastName)
    {
        if(String.IsNullOrWhiteSpace(firstName) || String.IsNullOrWhiteSpace(lastName))
        {
            throw new Exception("The name is invalid!");
        }
    }
}
```

```
public class EmailService
{
    public void Validate(string email)
    {
        if (!email.Contains("@") || !email.Contains("."))
        {
            throw new Exception("Email is not valid!!");
        }
    }
}
```

Single-Responsibility principle

"A class should have only a single responsibility (one reason to change)"

```
public class InvitationService
{
    UserNameService _userNameService;
    EmailService _emailService;

    public InvitationService(UserNameService userNameService, EmailService emailService)
    {
        _userNameService = userNameService;
        _emailService = emailService;
    }
    public void SendInvite(string email, string firstName, string lastName)
    {
        _userNameService.Validate(firstName, lastName);
        _emailService.Validate(email);
        SmtpClient client = new SmtpClient();
        client.Send(new MailMessage("sitename@invites2you.com", email) { Subject = "Please join me
    }
}
```

Single-Responsibility principle

"A class should have only a single responsibility (one reason to change)"

```
$shapes = array(
    new Circle(2),
    new Square(5),
    new Square(6)
);

$areas = new AreaCalculator($shapes);

echo $areas->output();
```

Single-Responsibility principle

"A class should have only a single responsibility (one reason to change)"

```
$shapes = array(
    new Circle(2),
    new Square(5),
    new Square(6)
);

$areas = new AreaCalculator($shapes);
$output = new SumCalculatorOutputter($areas);

echo $output->JSON();
echo $output->HAML();
echo $output->HTML();
echo $output->JADE();
```

Open-Closed principle

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

Open-Closed principle

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

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

```
public class Circle
{
    public double Radius { get; set; }
}
```

```
public class CombinedAreaCalculator
{
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle)shape;
                area += rectangle.Width * rectangle.Height;
            }
        }
        return area;
    }
}
```

Open-Closed principle

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

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

```
public class Circle
{
    public double Radius { get; set; }
}
```

```
public class CombinedAreaCalculator
{
    public double Area(object[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            if (shape is Rectangle)
            {
                Rectangle rectangle = (Rectangle)shape;
                area += rectangle.Width * rectangle.Height;
            }
            if (shape is Circle)
            {
                Circle circle = (Circle)shape;
                area += (circle.Radius * circle.Radius) * Math.PI;
            }
        }

        return area;
    }
}
```

Open-Closed principle

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

```
public abstract class Shape
{
    public abstract double Area();
}
```

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width * Height;
    }
}
```

```
public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius * Radius * Math.PI;
    }
}
```

```
public class Triangle : Shape
{
    public double Height { get; set; }
    public double Width { get; set; }
    public override double Area()
    {
        return Height * Width * 0.5;
    }
}
```

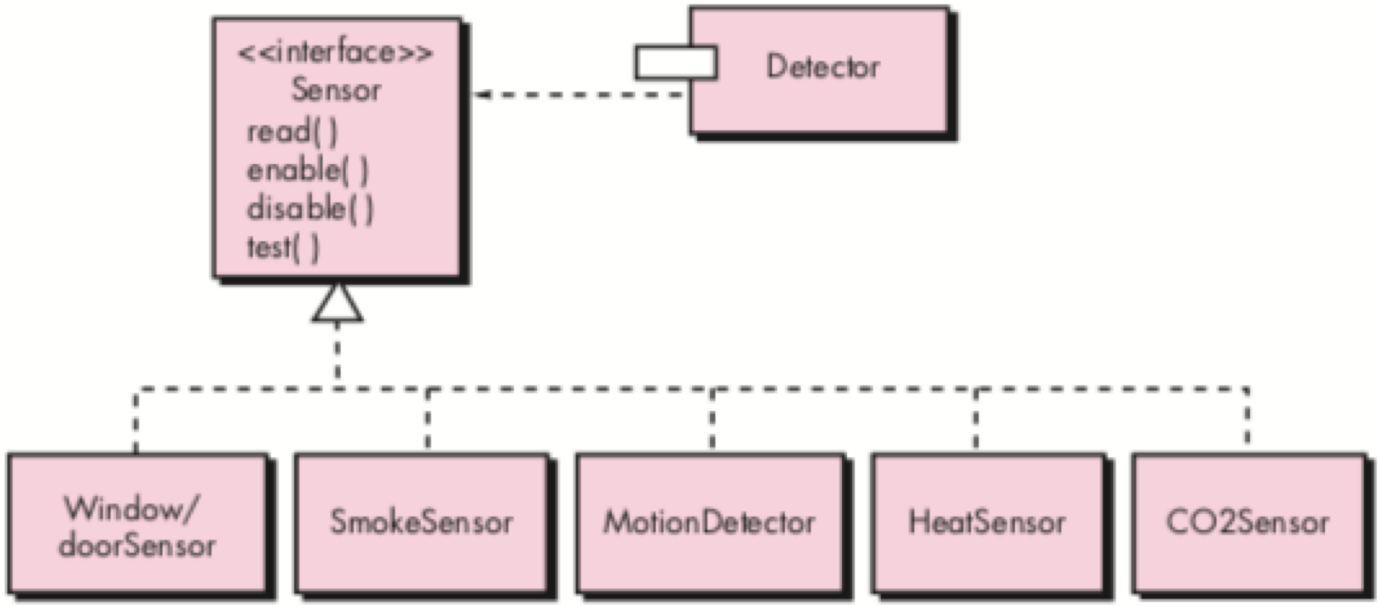
Open-Closed principle

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

```
public class CombinedAreaCalculator
{
    public double Area(Shape[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Area();
        }
        return area;
    }
}
```

Open-Closed principle

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



Liskov Substitution principle

"Subclasses should be substituable for their base classes"

Liskov Substitution principle

"Subclasses should be substituable for their base classes"

```
class Bird {
  public void fly(){}
  public void eat(){}
}
class Crow extends Bird {}
class Ostrich extends Bird{
  fly(){
    throw new UnsupportedOperationException();
  }
}
```

```
public BirdTest{
  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird());
    birdList.add(new Crow());
    birdList.add(new Ostrich());
    letTheBirdsFly ( birdList );
  }
  static void letTheBirdsFly ( List<Bird> birdList ){
    for ( Bird b : birdList ) {
      b.fly();
    }
  }
}
```

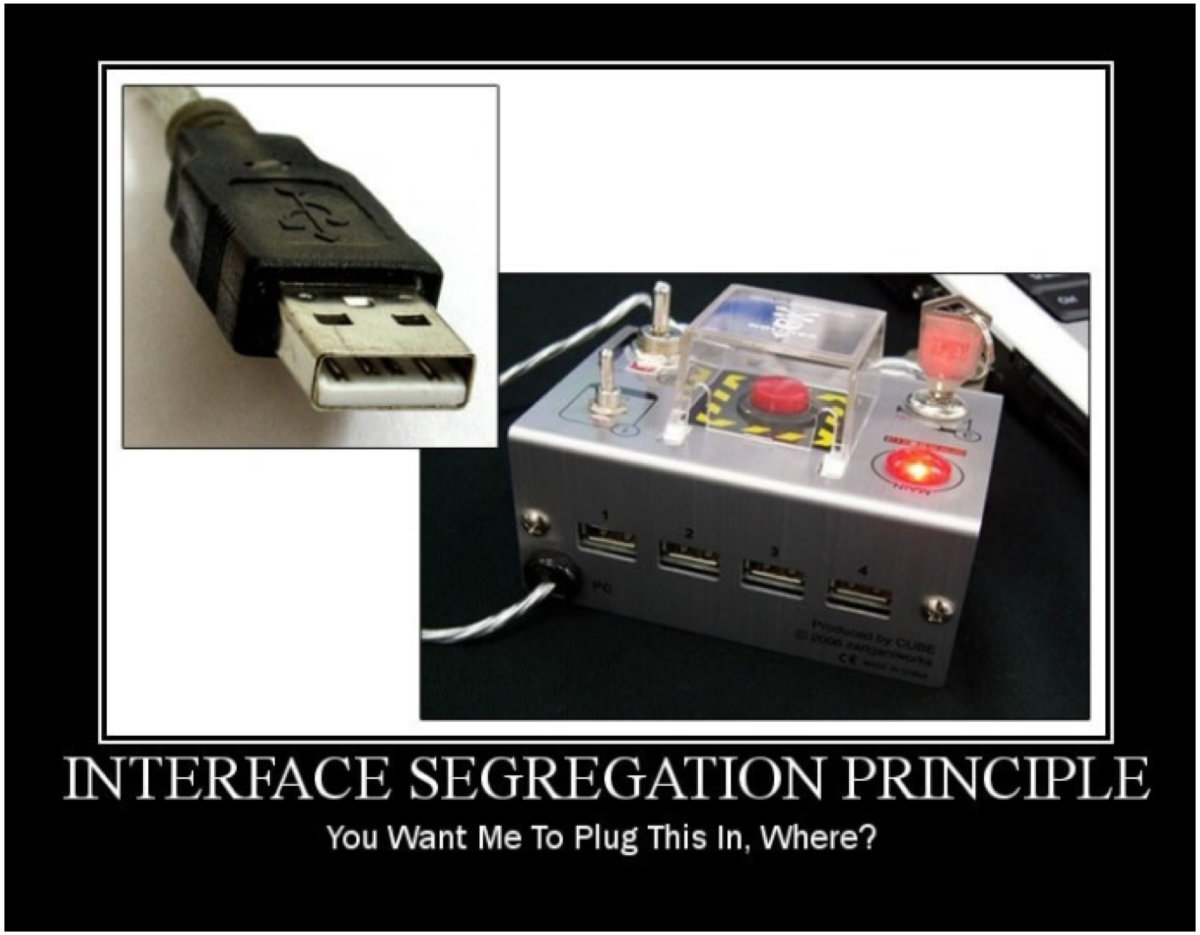
Liskov Substitution principle

"Subclasses should be substituable for their base classes"

```
class Bird{
  public void eat(){}
}
class FlightBird extends Bird{
  public void fly()()
}
class NonFlight extends Bird{}
```

Interface Segregation principle

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



Interface Segregation principle

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

```
interface ShapeInterface {
    public function area();
    public function volume();
}
```

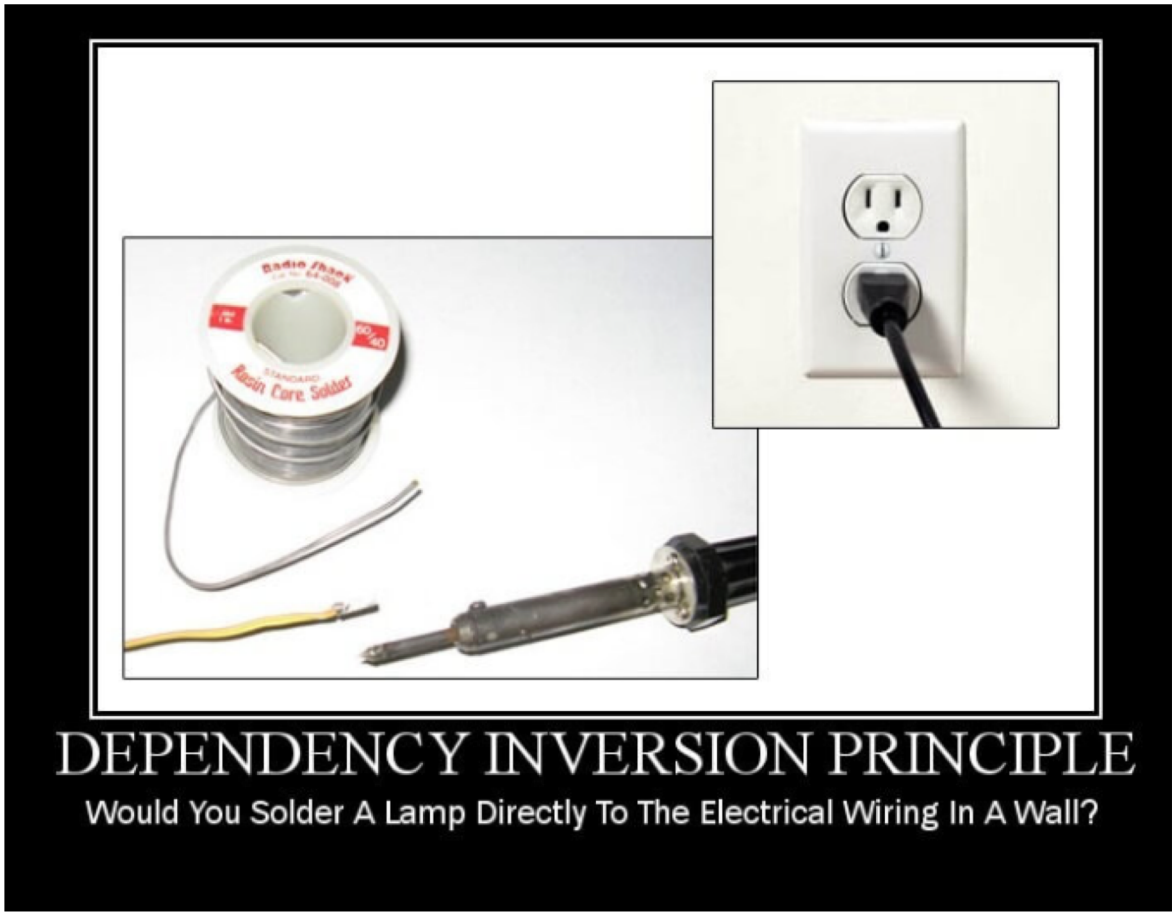
Interface Segregation principle

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

```
interface ShapeInterface {
    public function area();
}

interface SolidShapeInterface {
    public function volume();
}

class Cuboid implements ShapeInterface, SolidShapeInterface {
    public function area() {
        // calculate the surface area of the cuboid
    }

    public function volume() {
        // calculate the volume of the cuboid
    }
}
```

Dependency Inversion principle

"Depend on abstractions, do not depend on concretions"



Dependency Inversion principle

"Depend on abstractions, do not depend on concretions"

```
public class Email
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendEmail()
    {
        //Send email
    }
}
```

```
public class SMS
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendSMS()
    {
        //Send sms
    }
}
```

```
public class Notification
{
    private Email _email;
    private SMS _sms;
    public Notification()
    {
        _email = new Email();
        _sms = new SMS();
    }

    public void Send()
    {
        _email.SendEmail();
        _sms.SendSMS();
    }
}
```

Dependency Inversion principle

"Depend on abstractions, do not depend on concretions"

```
public interface IMessage
{
    void SendMessage();
}
```

```
public class Email : IMessage
{
    public string ToAddress { get; set; }
    public string Subject { get; set; }
    public string Content { get; set; }
    public void SendMessage()
    {
        //Send email
    }
}
```

```
public class SMS : IMessage
{
    public string PhoneNumber { get; set; }
    public string Message { get; set; }
    public void SendMessage()
    {
        //Send sms
    }
}
```

Dependency Inversion principle

"Depend on abstractions, do not depend on concretions"

```
public class Notification
{
    private ICollection<IMessage> _messages;

    public Notification(ICollection<IMessage> messages)
    {
        this._messages = messages;
    }
    public void Send()
    {
        foreach(var message in _messages)
        {
            message.SendMessage();
        }
    }
}
```

Referencias

- https://exceptionnotfound.net/tag/solid-principles/
- https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- https://www.javacodegeeks.com/2011/11/solid-liskov-substitution-principle.html
- https://www.youtube.com/watch?v=TAVn7s-kO9o

Actividad 2

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

Clase 2
Diseño de Componentes

IIC2113 – Diseño Detallado de Software

Rodrigo Saffie

rasaffie@uc.cl

24 de agosto de 2018