IIC2113 - Diseño Detallado de Software

CLASE 3 - BUENAS PRÁCTICAS DE DESARROLLO

Pontificia Universidad Católica de Chile
2020-2

Índice

01. Buenas prácticas

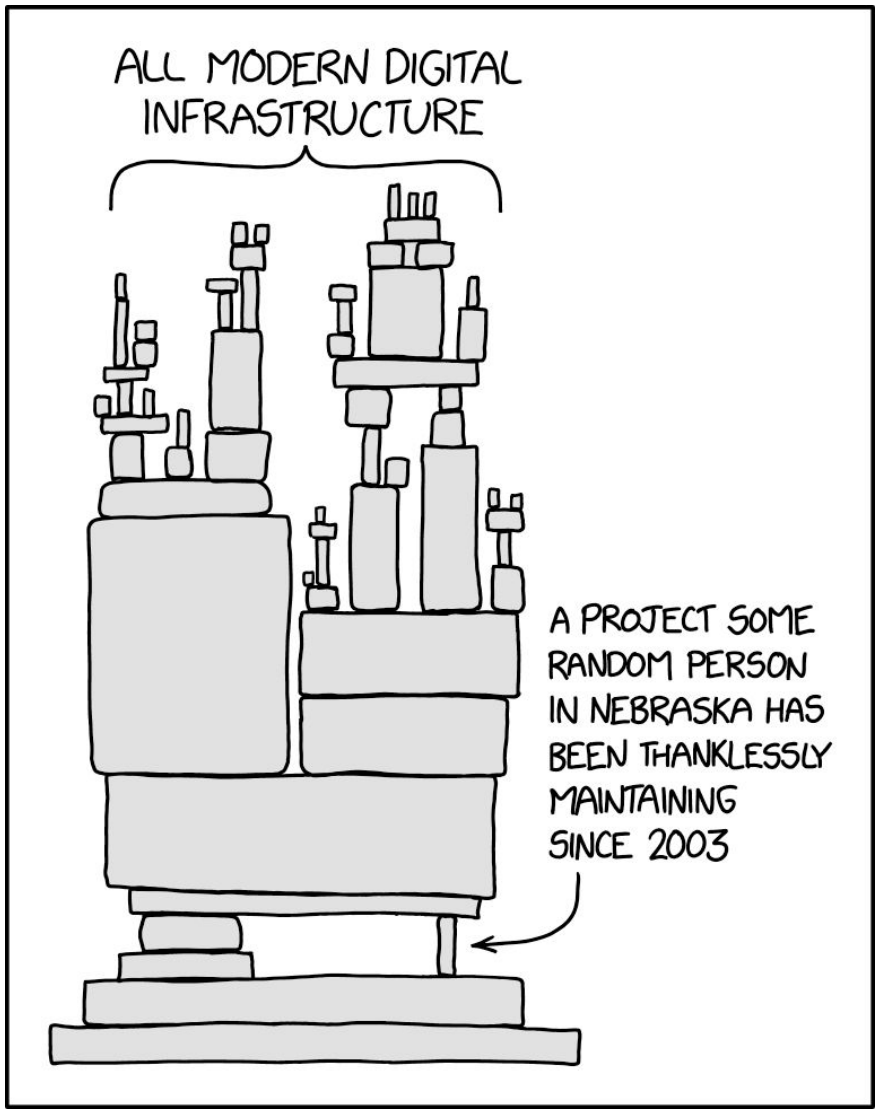
¿A qué nos referimos con “Buenas Prácticas”?

¿Buenas prácticas?

"Small things matters"

Es un dicho ampliamente popular. Sin embargo no le quita lo veraz.

La práctica en los **detalles** permite al **profesional** mayor dominio y confianza en su trabajo. Por otra parte un fallo en construcción genera un desencanto por el trabajo en general (Robert C. Martin, 2008).



https://xkcd.com/2347/

Total productive maintenance (TPM) - 5S principles

TPM: Filosofía originaria de Japón, el cual se enfoca en la eliminación de pérdidas asociadas con la detención, calidad y costes en los procesos de producción industrial.
¿Cómo se relaciona esto con el código?

Seiri **(organización)**

Saber dónde usando **convenciones de nombre**.

Seiso **(limpieza, brillar)**

Mantén el espacio de trabajo libre de desechos. Si una línea de código no aporta, **eliminala**.

Seiton **(ordenar, sistematizar)**

Una pieza de código debe estar donde esperas encontrarlo. Si no es así, ***refactor***.

Shutsuke **(disciplina)**

Tener la **disciplina** para seguir las prácticas, **reflejar** en el trabajo, y voluntad de **adaptarse**.

Seiketsu **(estandarización)**

Como **grupo** se llega a un **consenso** acerca de cómo mantener el espacio limpio

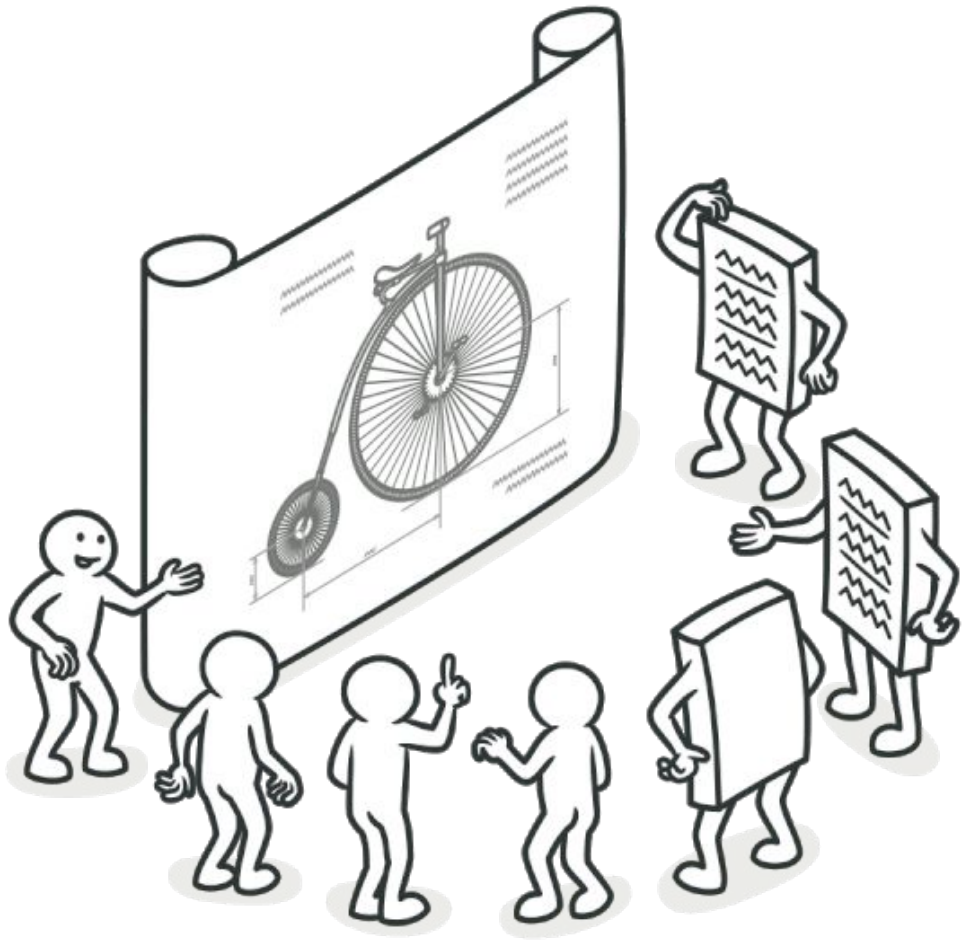
TPM apunta a que no siempre hay que apuntar a producir a velocidad óptima → sino que a crear máquinas que se mantengan mejor.

¿A qué nos referimos con buenas prácticas?

A todas aquellas prácticas que nos permitan crear **código mantenible**.

Buenas prácticas

- El código mantenible es igual de importante que el código ejecutable.
- Se debe apuntar a criterios de robustez, extensibilidad y mantenibilidad.
- No es lo mismo escribir **código limpio** para un proyecto sencillo, que para un proyecto complejo compuesto de varios componentes obligados a cooperar.



02. Principios SOLID

Principios SOLID

- Propuestos por Robert C. Martin (cerca del 2000), en el libro “Agile Principles, Patterns and Practices in C#”.
- Constituyen un set de estándares de alto nivel para construir código de calidad en lenguajes orientados a objetos.
- Estos principios son:
    - **S**: Single-responsibility principle
    - **O**: Open-closed principle
    - **L**: Liskov substitution principle
    - **I**: Interface segregation principle
    - **D**: Dependency Inversion principle

SOLID - Single-responsibility principle

"***A class should have one and only one reason to change, meaning that a class should have only one job***"

- Una clase debe tener solo una responsabilidad.
- Relacionado con el concepto de **cohesión** (alta cohesión y bajo acoplamiento).
- Algunos errores comunes:
  - Logging: generalmente los logs deberían estar encapsulados fuera de la clase en cuestión.
  - Persistencia (lectura y escritura): lógicas de serialización y deserialización deberían estar de forma aparte, como en un *Factory*.
  - Validación de formularios: debería estar encapsulado bajo su propia lógica de validación.

SOLID - Single-responsibility principle

```csharp
public class Rectangle {
  public int X { get; }
  public int Y { get; }
  public int Height { get; }
  public int Width { get; }

  public Rectangle(int x, int y, int height, int width) {
    X = x;
    Y = y;
    Height = height;
    Width = width;
  }

  public void Draw() {
    // Logic for drawing...
  }
}
```

```ruby
class Rectangle
  attr_accessor :x, :y, :height, :width

  def initialize(x, y, height, width)
    @x = x
    @y = y
    @height = height
    @width = width
  end

  def draw
    # Logic for drawing...
  end
end
```

El ejemplo **no cumple** el SRP, pues dispone de dos responsabilidades:

- Proveer el modelo para la representación geométrica de un rectángulo.
- Dibujar el rectángulo en una interfaz gráfica.

SOLID - Open-Closed Principle

"***Objects or entities should be open for extension, but closed for modification***"

- Establece que una clase debiese estar abierta a extensión y cerrada a modificación.
- No debiera modificarse el código interno de una clase una vez que está creada y siendo utilizada.
- En lenguajes de tipo estático, generalmente se exponen interfaces para adaptar los componentes:
    - A pesar de lo anterior, seguir "*fool me once, shame on you; fool me twice, shame on me*" -> no es necesario abstraer todo de inmediato, prefiere refactorizar cuando sea necesario.

SOLID - Open-Closed Principle

```
1   public class Renderer {
2     public void draw(Shape shape) {
3       if (shape is Circle) {
4         // Draw circle
5       }
6       else if (shape is Square) {
7         // Draw rectangle
8       }
9       //  ...
10    }
11  }
```

```
1   class Renderer
2     def draw(shape)
3       if shape.is_a? Circle
4         # Draw circle
5       elsif shape.is_a? Square
6         # Draw rectangle
7       end
8       #  ...
9     end
10  end
```

El ejemplo **no cumple** el OCP, dado que si a futuro yo quisiera incorporar un nuevo tipo de Shape (por ejemplo Triangle), me vería obligado a modificar la implementación interna de Renderer (agregar una nueva condición if/else).

SOLID - Liskov Substitution Principle

"***Subclasses should be substitutable for their base classes***"

- Establece que todo subtipo debe ser posible sustituirlo por un supertipo sin introducir comportamiento errático o anti-intuitivo en el programa.
- Suele darse cuando:
  - Se abusa de la herencia.
  - Se sobre-escribe un método en una clase derivada con otro que no hace nada o lanza una excepción.
  - Dada una clase padre con un método con cierta lógica, sobreescribirlo en una clase derivada con otro método que haga algo completamente distinto y nunca llame al método padre.

SOLID - Liskov Substitution Principle

```
1   public class Shape {}
2
3   public class Rectangle : Shape {
4     public virtual int Height { get; set; }
5     public virtual int Width { get; set; }
6   }
7
8   public class Square : Rectangle {
9     public override int Height {
10      get ⇒ base.Height;
11      set {
12        base.Height = value;
13        base.Width = value;
14      }
15    }
16
17    public override int Width {
18      get ⇒ base.Width;
19      set {
20        base.Height = value;
21        base.Width = value;
22      }
23    }
24  }
```

```
1   class Shape; end
2
3   class Rectangle < Shape
4     attr_reader :height, :width
5
6     def initialize(height, width)
7       @height = height
8       @width = width
9     end
10  end
11
12  class Square < Rectangle
13    def initialize(side)
14      super(side, side)
15    end
16
17    def height=(h)
18      @height = h
19      @width = h
20    end
21
22    def width=(w)
23      @width = w
24      @height = w
25    end
26  end
```

El ejemplo **no cumple** el LSP dado que si yo tengo un Square, le hago un casting a Rectangle, y luego alguien setea las dimensiones del Rectangle en un nuevo valor, resultaría incomprensible el ser incapaz de fijar dimensiones distintas para altura y ancho.

SOLID - Liskov Substitution Principle

```
1   public class Shape {}
2
3   public class Rectangle : Shape {
4     public virtual int Height { get; set; }
5     public virtual int Width { get; set; }
6   }
7
8   public class Square : Rectangle {
9     public override int Height {
10      get => base.Height;
11      set {
12        base.Height = value;
13        base.Width = value;
14      }
15    }
16
17    public override int Width {
18      get => base.Width;
19      set {
20        base.Height = value;
21        base.Width = value;
22      }
23    }
24  }
```

```
1   class Shape; end
2
3   class Rectangle < Shape
4     attr_reader :height, :width
5
6     def initialize(height, width)
7       @height = height
8       @width = width
9     end
10  end
11
12  class Square < Rectangle
13    def initialize(side)
14      super(side, side)
15    end
16
17    def height=(h)
18      @height = h
19      @width = h
20    end
21
22    def width=(w)
23      @width = w
24      @height = w
25    end
26  end
```

Aparte: este ejemplo muestra una de las claras diferencias entre C# y Ruby.
C# es estático y fuertemente tipado, mientras que Ruby es dinámico y refuerza el duck typing.
Veremos esto en detalle en la clase de paradigmas de programación.

SOLID - Interface Segregation Principle

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

- Desventaja de interfaces con demasiados métodos (*fat interfaces*).
- Clases que implementan interfaces no debieran estar forzadas a requerir implementaciones de métodos irrelevantes para ellas.
- Muchas veces resulta deseable descomponer las interfaces grandes en varias interfaces más pequeñas.

SOLID - Interface Segregation Principle

```
1  public interface ISurface {
2    double GetSurface();
3    double Eccentricity();
4  }
5
6  public class Oval : Shape, ISurface {
7    public double GetSurface() {
8      return Math.PI * (Height / 2) * (Width / 2);
9    }
10
11   public double Eccentricity() {
12     if (Height > Width) {
13       return Math.Sqrt(1 - Math.Pow(Width / 2, 2) / Math.Pow(Height / 2, 2));
14     } else {
15       return Math.Sqrt(1 - Math.Pow(Height / 2, 2) / Math.Pow(Width / 2, 2));
16     }
17   }
18 }
19
20 public class Rectangle : Shape, ISurface {
21   public double GetSurface() {
22     return Height * Width;
23   }
24
25   public double Eccentricity() {
26     return 0;
27   }
28 }
```

El ejemplo **no cumple** el ISP dado que el concepto de *Eccentricity* no está definido para cualquier tipo de superficie y no tiene sentido forzar que otras figuras también lo definan.

SOLID - Dependency Inversion Principle

"***Depend on abstractions. Do not depend on concretions***"

- Establece que módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones.
- Principio orientado a facilitar el proceso de testing y extensión en las clases que definamos.
- Una forma para cumplir con el principio es usando el patrón de diseño Inyección de Dependencia (*Dependency Injection*):
    - En vez de crear el objeto en la clase, se le suministra el objeto a través del constructor, una propiedad o un método (de ahí Injection).

SOLID - Dependency Inversion Principle

```java
public class NotificationHandler {
  public Logger Logger = new Logger();
  public Mailer Mailer = new Mailer();

  public void notifyEvent() {
    try {
      Mailer.sendEmail("...");
      Logger.log("Mail sent");
    }
    catch (Exception e) {
      Logger.log("Failed to sent email");
      Logger.log(e.ToString());
    }
  }
}
```

```ruby
class NotificationHandler
  @@logger = Logger.new
  @@mailer = Mailer.new

  def self.notify_event
    begin
      @@mailer.send_email('...')
      @@logger.log('Mail sent')
    rescue Exception ⇒ ex
      @@logger.log('Failed to sent email')
      @@logger.log(ex)
    end
  end
end
```

El ejemplo **no cumple** el DIP dado que NotificationHandler (módulo de alto nivel) establece una dependencia directa con Logger y Mailer.

03. Métricas de calidad

¿Qué es la calidad?

Garvin (1984) define la calidad según vistas:

- **Vista trascendental**: la calidad se percibe, pero no se puede explicar.
- **Vista del usuario**: la calidad en base a los objetivos del usuario final.
- **Vista del productor**: la calidad según las especificaciones del producto.
- **Vista del producto**: la calidad en función de lo que hace el producto.
- **Vista del valor**: la calidad en base a lo que está dispuesto a pagar un consumidor.

Pressman (2009) lo define como:

"Un desarrollo de software efectivo, aplicado de una manera que crea un producto útil que provee valor cuantificable para aquellos que lo producen y aquellos que lo utilizan"

La calidad depende varios factores

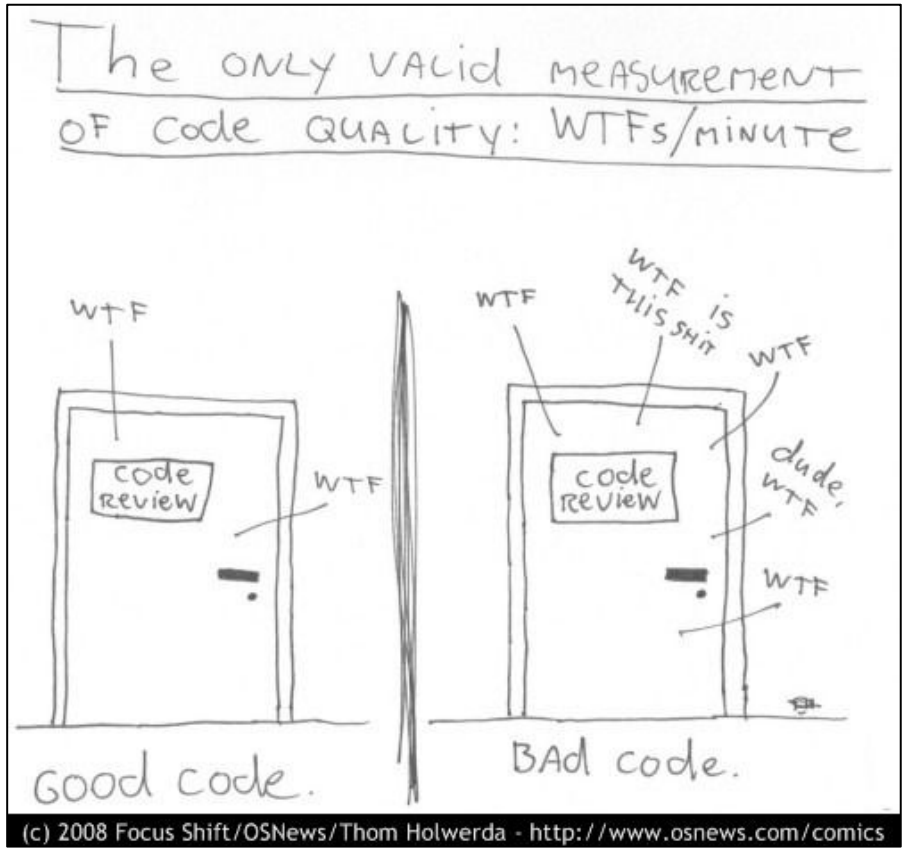
Factores de calidad definidos por McCall (1977):

- Revisión de producto:
  - Mantenibilidad (maintainability)
  - Flexibilidad (flexibility)
  - Testabilidad (testability)
- Transición del producto:
  - Portabilidad (portability)
  - Reusabilidad (reusability)
  - Interoperabilidad (interoperability)
- Operación del producto:
  - Correctitud (correctness)
  - Confiabilidad (reliability)
  - Usabilidad (usability)
  - Integridad (integrity)
  - Eficiencia (efficiency)
  - Interoperabilidad (interoperability)

Métricas de calidad

Las métricas de calidad son valores objetivos, sencillos y fáciles de computar, que además cumples con ser:

- Consistentes
- Con unidades de medición expresivas
- Independientes del lenguaje de programación
- Reflejan recomendaciones para mejorar la calidad del software



2: Ilustración de Thom Holwerda, sitio web de https://www.osnews.com/comics/.

Complejidad ciclomática (Cyclomatic Complexity)

- Es una métrica basada en el cálculo del número de **caminos independientes** que tiene el código.
- Decimos que un método aumenta en complejidad a medida que incluye más y más bifurcaciones, loops y anidaciones.
- Apunta a la testabilidad y complejidad.
- La métrica **Weighted Method per Class** (WMC) es la suma de los CYC en una clase. Es un indicador predictivo del esfuerzo necesario para mantener y extender una clase.

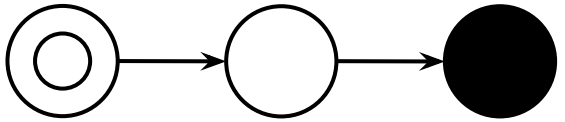
La complejidad ciclomática (CYC) se calcula como:

$$CYC = E - N + 2 * P$$

Donde:
E: Aristas (edges), transferencias de control (bifurcaciones, loops, anidaciones).
N: Nodos, son los bloques de código o statements.
P: Componentes conectados, como subrutinas o llamadas a programas. Si evaluamos un solo método entonces P = 1.

Complejidad ciclomática - Ejemplo

```ruby
def print_hello
  puts 'hello'
  puts 'hello'
  puts 'hello'
  puts 'hello'
end
```



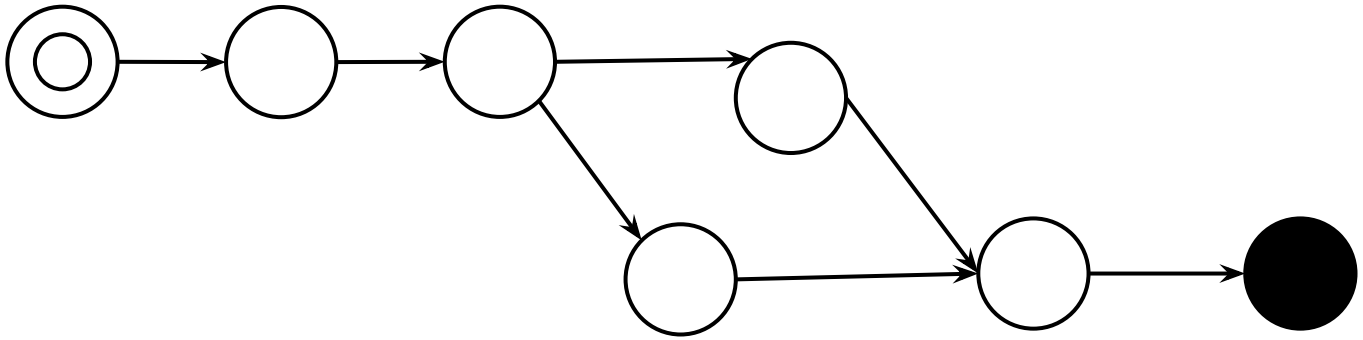
$$CYC = E - N + 2 * P$$

$$CYC = 2 - 3 + 2 * 1$$

$$CYC = 1$$

Complejidad ciclomática - Ejemplo

```ruby
def stuff(b, c)
  a = 10
  if b < c
    a = b
  else
    a = c
  end
  puts a
  puts b
  puts c
end
```



$$CYC = E - N + 2 * P$$

$$CYC = 7 - 7 + 2 * 1$$

$$CYC = 2$$

Complejidad ciclomática

- Generalmente, es la métrica que usan para medir la complejidad los **linters** más comunes:
  - [Javascript] Eslint -> https://eslint.org/docs/rules/complexity
  - [Ruby] Rubocop -> https://docs.rubocop.org/rubocop/cops_metrics.html#metricscyclomaticcomplexity
  - [C#] FxCop -> https://docs.microsoft.com/en-us/visualstudio/code-quality/ca1502?view=vs-2019
  - [Swift] SwiftLint -> https://realm.github.io/SwiftLint/cyclomatic_complexity.html

ABC Size

- También mide complejidad, se basa en:
  - **Asignaciones**:
    - operadores =, +=, ++, --, etc.-
  - **Ramas**: llamadas a otra parte del programa:
    - funciones, goto, new.
  - **Condicionales**:
    - <, >, >=, !=, etc.-
    - if-else, case, default

El ABC Size se calcula como:

$$ABC_{size} = \sqrt{Asignaciones^2 + Ramas^2 + Condicionales^2}$$

Depth of Inheritance Tree (DIT)

- Es la máxima profundidad de una clase en su árbol de jerarquía de clases:
  - Una clase que no tiene ninguna superclase tiene DIT = 1
  - Los hijos de una clase tienen DIT = 2
  - Los hijos de los hijos de una clase tienen DIT = 3
- A medida que aumenta el DIT se vuelve más complicado entender el comportamiento de una clase dado que requiere entender y visualizar más y más capas de herencia. Hay un trade-off con la reutilización del código: mayor repetición pero mayor complejidad.

Number of Children (NOC)

- Cantidad de subclases directas de una clase.
  - Número de hijos de una clase.
- Refleja el nivel de abstracción del diseño.
- NOC es un indicador de riesgo.

Coupling Between Objects (CBO)

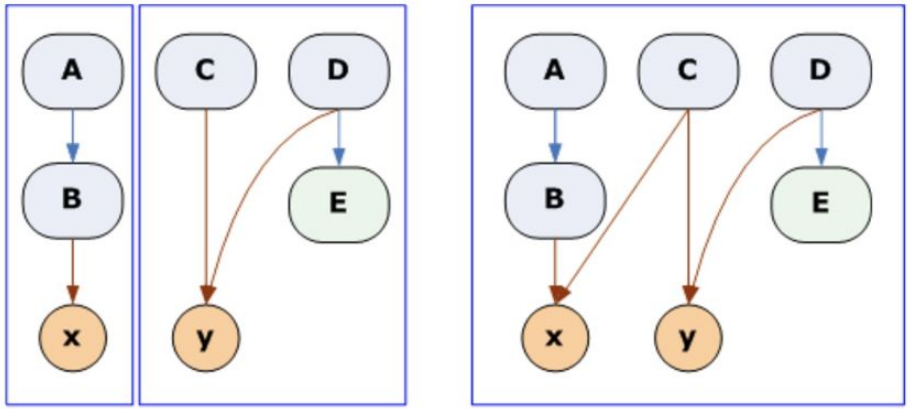
- Cantidad de clases con las que colabora una clase.
  - Decimos que hay una relación de asociación entre A y B si A invoca un método o atributo de B o vice-versa.
  - También consideramos el número de interfaces a los que una clase haga referencia.
- Altos niveles reflejan alto acoplamiento, lo que dificulta modificar el código.

Response for Class (RFC)

- Cantidad de métodos únicos invocados desde una clase (propios y los que utiliza directamente)
  - RFC = Nº métodos totales - Nº métodos privados
- Altos niveles reflejan alta complejidad, lo que dificulta entender y probar el código (complejidad y testabilidad).

LCOM: Lack of Cohesion Of Methods

- Cantidad de grupos de métodos que acceden a 1 o más atributos en común de una clase.
- Altos niveles reflejan poca cohesión, lo que refleja que la responsabilidad de una clase no está bien definida.
- Una debilidad de LCOM es es comparable entre números de métodos



05. Próxima Clase

Próxima clase

- Buenas prácticas Parte 2
  - Code smells
  - Refactoring
  - Testing

Clase 1 - Motivación

Bibliografía

- R.S. Pressman, Software Engineering: A Practitioner's Approach.
- S. Chidamber, C. Kemerer. Transactions on Software Engineering.
- R. C. Martin, Agile Principles, Patterns, and Practices in C#.

IIC2113 - Diseño Detallado de Software

Fernanda Sepúlveda - mfsepulveda@uc.cl

Pontificia Universidad Católica de Chile
2020-2