

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

Clase 06: Clean Code (Cap. 6)

Francisco Gazitúa Requena (fco_gazitua_requena@uc.cl)

IIC2113 Diseño Detallado de Software

21 de Agosto, 2025

Objects and Data Structures



Objects and Data Structures

"There is a reason that we keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?"

– Robert C. Martin.

Data Abstraction

Opción 1:

```
2 public class Point {
3     public double x;
4     public double y;
5 }
```

Opción 2:

```
2 public interface Point {
3     double getX();
4     double getY();
5     void setCartesian(double x, double y);
6     double getR();
7     double getTheta();
8     void setPolar(double r, double theta);
9 }
```

¿Cuál implementación es mejor? ¿por qué?

Data Abstraction

Opción 1:

```
2 public interface Vehicle {
3     double getFuelTankCapacityInGallons();
4     double getGallonsOfGasoline();
5 }
```

Opción 2:

```
2 public interface Vehicle {
3     double getPercentFuelRemaining();
4 }
```

¿Cuál interfaz es mejor? ¿por qué?

Data Abstraction

Lección 1: "*We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters.*"

Data Abstraction

Lección 1: "*We do not want to expose the details of our data. Rather we want to express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters. Serious thought needs to be put into the best way to represent the data that an object contains. The worst option is to blithely add getters and setters.*"

```
2 public interface Vehicle {
3     double getPercentFuelRemaining();
4 }
```

Data/Object Anti-Symmetry

Otro ejemplo: Queremos calcular al área de figuras geométricas.

```
1  public class Square {
2      public double side;
3  }
4
5  public class Rectangle {
6      public double height;
7      public double width;
8  }
9
10 public class Circle {
11     public double radius;
12 }
```

Data/Object Anti-Symmetry

Otro ejemplo: Queremos calcular al área de figuras geométricas.

```
14 public class Geometry {
15     public double Area(Object shape) {
16         if (shape is Square) {
17             Square s = (Square)shape;
18             return s.side * s.side;
19         }
20         else if (shape is Rectangle) {
21             Rectangle r = (Rectangle)shape;
22             return r.height * r.width;
23         }
24         else if (shape is Circle) {
25             Circle c = (Circle)shape;
26             return Math.PI * c.radius * c.radius;
27         }
28         throw new NoSuchShapeException();
29     }
30 }
```

Data/Object Anti-Symmetry

¿Ven algún problema con el ejemplo anterior?

Data/Object Anti-Symmetry

¿Ven algún problema con el ejemplo anterior?

El tamaño del método explota.

Data/Object Anti-Symmetry

¿Ven algún problema con el ejemplo anterior?

El tamaño del método explota.

```
1 public abstract class Shape {
2     public abstract double Area();
3 }
```

Data/Object Anti-Symmetry

¿Ven algún problema con el ejemplo anterior?

El tamaño del método explota.

```
1  public abstract class Shape {
2      public abstract double Area();
3  }
```

Luego cada figura es una clase que hereda de Shape.

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

Data/Object Anti-Symmetry

¿Ven algún problema con el ejemplo anterior?

El tamaño del método explota.

```
1  public abstract class Shape {
2      public abstract double Area();
3  }
```

Luego cada figura es una clase que hereda de Shape.

```
13 public class Rectangle: Shape {
14     private double height;
15     private double width;
16
17     public override double Area() {
18         return height * width;
19     }
20 }
```

Data/Object Anti-Symmetry

¿Ven algún problema con el ejemplo anterior?

El tamaño del método explota.

```
1  public abstract class Shape {
2      public abstract double Area();
3  }
```

Luego cada figura es una clase que hereda de Shape.

```
22 public class Circle: Shape {
23     private double radius;
24
25     public override double Area() {
26         return Math.PI * radius * radius;
27     }
28 }
```

Data/Object Anti-Symmetry

Estructuras de datos: Las estructuras de datos exponen sus datos. Sus funciones, de tenerlas, son simples y permiten obtener y editar sus datos.

```
1 public class Square {
2     public double side;
3 }
```

Data/Object Anti-Symmetry

Estructuras de datos: Las estructuras de datos exponen sus datos. Sus funciones, de tenerlas, son simples y permiten obtener y editar sus datos.

```
1  public class Square {
2      public double side;
3  }
```

Objetos: Los objetos esconden sus datos detrás de abstracciones y exponen funciones que operan sobre ellos.

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

Data/Object Anti-Symmetry

Solución 1: Estructuras simples. El área se calcula en un método con un `if` por figura.

```
1  public class Square {
2      public double side;
3  }
```

Solución 2: Objetos complejos. El área se calcula dentro de cada objeto.

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

Data/Object Anti-Symmetry

Solución 1: Estructuras simples. El área se calcula en un método con un `if` por figura.

```
1  public class Square {
2      public double side;
3  }
```

Solución 2: Objetos complejos. El área se calcula dentro de cada objeto.

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

¿Cuál opción es mejor?

Data/Object Anti-Symmetry

La respuesta es “*depende*”.

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

¿Qué pasa si queremos agregar una nueva figura?

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

¿Qué pasa si queremos agregar una nueva figura?

- No hay que editar nada, solo crear una nueva clase que herede de `Shape`.

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

¿Qué pasa si queremos agregar una nueva figura?

- No hay que editar nada, solo crear una nueva clase que herede de `Shape`.

¿Qué pasa si queremos agregar la función `Perimeter()`?

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
5  public class Square: Shape {
6      private double side;
7
8      public override double Area() {
9          return side*side;
10     }
11 }
```

¿Qué pasa si queremos agregar una nueva figura?

- No hay que editar nada, solo crear una nueva clase que herede de `Shape`.

¿Qué pasa si queremos agregar la función `Perimeter()`?

- Debemos editar **todas** las clases que hereden de `Shape` y agregar `Perimeter()`.

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
1 public class Square {
2     public double side;
3 }
```

¿Qué pasa si queremos agregar una nueva figura?

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
1 public class Square {
2     public double side;
3 }
```

¿Qué pasa si queremos agregar una nueva figura?

- Hay que editar el método `Area(Object shape)` y agregar un nuevo `if`.

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
1 public class Square {
2     public double side;
3 }
```

¿Qué pasa si queremos agregar una nueva figura?

- Hay que editar el método `Area(Object shape)` y agregar un nuevo `if`.

¿Qué pasa si queremos agregar la función `Perimeter()`?

Data/Object Anti-Symmetry

La respuesta es "*depende*".

```
1 public class Square {
2     public double side;
3 }
```

¿Qué pasa si queremos agregar una nueva figura?

- Hay que editar el método `Area(Object shape)` y agregar un nuevo `if`.

¿Qué pasa si queremos agregar la función `Perimeter()`?

- No debemos editar **ninguna** figura, solo agregar un nuevo método `Perimeter(Object shape)`.

Data/Object Anti-Symmetry

Lección 2: "*Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.*"

Data/Object Anti-Symmetry

Lección 2: "*Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.*"

Corollary: "*In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate.*"

Data/Object Anti-Symmetry

Lección 2: "*Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.*"

Corollary: "*In any complex system there are going to be times when we want to add new data types rather than new functions. For these cases objects and OO are most appropriate. On the other hand, there will also be times when we'll want to add new functions as opposed to data types. In that case procedural code and data structures will be more appropriate.*"

¿Y si tenemos un híbrido y así tenemos lo mejor de los dos mundos? 🤔

Objects and Data Structures: Data/Object Anti-Symmetry

```
public class Square: Shape {
    public double side;

    public override double Area() {
        return side*side;
    }
}

public class Geometry {
    public double Perimeter(Object shape) {
        if (shape is Square) {
            Square s = (Square)shape;
            return 4 * s.side;
        }
        /* ... etc */
    }
}
```

Objects and Data Structures: Data/Object Anti-Symmetry

¿Son estas clases objetos?

```
public abstract class Unit
{
    public int MaxHP { get; set; }
    public int CurrentHP { get; set; }
    public int MaxMP { get; set; }
    public int CurrentMP { get; set; }
    public int Str { get; set; }
    public int Skl { get; set; }
    public int Mag { get; set; }
    public int Spd { get; set; }
    public int Lck { get; set; }
    public abstract Option[] Options { get; }
}
```

```
public Samurai : Unit
{
    public override Option[] Options { get; } = new[]
        { Option.Attack, Option.Shoot, Option.UseSkill, Option.SamuraiSummon, Option.Pass, Option.GiveUp
      };
}
```

```
public Monster : Unit
{
    public override Option[] Options { get; } = new[]
        { Option.Attack, Option.UseSkill, Option.MonsterSummon, Option.Pass };
}
```

Objects and Data Structures: Data/Object Anti-Symmetry

¿Es esta clase una estructura de datos?

```
public class TeamValidator
{
    public bool IsValidTeam(Unit[] team)
    {
        return IsSamuraiCountValid(team) &&
               IsMonsterCountValid(team) &&
               IsSamuraiSkillCountValid(team) &&
               !SamuraiHasDuplicatedSkills(team) &&
               !HasDuplicatedMonsters(team);
    }

    private bool IsSamuraiCountValid(Unit[] team){/* Código de validación */}

    private bool IsMonsterCountValid(Unit[] team){/* Código de validación */}

    private bool IsSamuraiSkillCountValid(Unit[] team){/* Código de validación */}

    private bool SamuraiHasDuplicatedSkills(Unit[] team){/* Código de validación */}

    private bool HasDuplicatedMonsters(Unit[] team){/* Código de validación */}
}
```

The Law of Demeter

The Law of Demeter

A method `f` of a class `C` should only call the methods of these:

- `C`.
- An object created by `f`.
- An object passed as an argument to `f`.
- An object held in an instance variable of `C`.

The Law of Demeter

The Law of Demeter

A method `f` of a class `C` should only call the methods of these:

- `C`.
- An object created by `f`.
- An object passed as an argument to `f`.
- An object held in an instance variable of `C`.

¿Qué ocurre en este caso?

```
2 outputDir = ctxt.GetOptions().GetScratchDir().GetAbsolutePath();
```

The Law of Demeter

The Law of Demeter

A method `f` of a class `C` should only call the methods of these:

- `C`.
- An object created by `f`.
- An object passed as an argument to `f`.
- An object held in an instance variable of `C`.

¿Qué ocurre en este caso?

```
2 outputDir = ctxt.GetOptions().GetScratchDir().GetAbsolutePath();
```

... este chico sabe demasiado.

The Law of Demeter

Paso 1: Mínimamente hay que eliminar el *train wreck*.

```
2 Options opts = ctxt.GetOptions();
3 File scratchDir = opts.GetScratchDir();
4 String outputDir = scratchDir.GetAbsolutePath();
```

The Law of Demeter

Paso 1: Mínimamente hay que eliminar el *train wreck*.

```
2 Options opts = ctxt.GetOptions();
3 File scratchDir = opts.GetScratchDir();
4 String outputDir = scratchDir.GetAbsolutePath();
```

Paso 2: Usar la ley de *Demeter*.

- `string outDir = ctxt.GetAbsolutePathOfScratchDirectoryOption();`

The Law of Demeter

Lección 3: "*There is a well-known heuristic called the Law of Demeter that says a module should not know about the innards of the objects it manipulates. As we saw, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.*"

The Law of Demeter

Lección 3: "*There is a well-known heuristic called the Law of Demeter that says a module should not know about the innards of the objects it manipulates. As we saw, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.*"

Ese es el problema principal con los *train wrecks*:

```
2 outputDir = ctxt.GetOptions().GetScratchDir().GetAbsolutePath();
```

The Law of Demeter

Lección 3: "*There is a well-known heuristic called the Law of Demeter that says a module should not know about the innards of the objects it manipulates. As we saw, objects hide their data and expose operations. This means that an object should not expose its internal structure through accessors because to do so is to expose, rather than to hide, its internal structure.*"

Ese es el problema principal con los *train wrecks*:

```
2 outputDir = ctxt.GetOptions().GetScratchDir().GetAbsolutePath();
```

Obs: La ley de Demeter solo aplica a objetos, no aplica a estructuras de datos.

Resumen

Resumen

Lección 1: Hay que pensar con cuidado qué dejamos público en nuestras clases. Todo método/atributo público nos quita libertad para modificar nuestras clases a futuro.

Resumen

Lección 1: Hay que pensar con cuidado qué dejamos público en nuestras clases. Todo método/atributo público nos quita libertad para modificar nuestras clases a futuro.

Lección 2: Un programador experimentado reconoce que herencia y polimorfismo no son siempre la solución apropiada a un problema.

Resumen

Lección 1: Hay que pensar con cuidado qué dejamos público en nuestras clases. Todo método/atributo público nos quita libertad para modificar nuestras clases a futuro.

Lección 2: Un programador experimentado reconoce que herencia y polimorfismo no son siempre la solución apropiada a un problema. Polymorfismo hace que sea fácil agregar nuevas clases sin cambiar funciones existentes.

Resumen

Lección 1: Hay que pensar con cuidado qué dejamos público en nuestras clases. Todo método/atributo público nos quita libertad para modificar nuestras clases a futuro.

Lección 2: Un programador experimentado reconoce que herencia y polimorfismo no son siempre la solución apropiada a un problema. Polymorfismo hace que sea fácil agregar nuevas clases sin cambiar funciones existentes. Pero si lo que necesitamos es agregar nuevas funciones sobre un conjunto fijo de clases entonces es mejor usar estructuras de datos simples con funciones que operen sobre ellas.

Resumen

Lección 1: Hay que pensar con cuidado qué dejamos público en nuestras clases. Todo método/atributo público nos quita libertad para modificar nuestras clases a futuro.

Lección 2: Un programador experimentado reconoce que herencia y polimorfismo no son siempre la solución apropiada a un problema. Polymorfismo hace que sea fácil agregar nuevas clases sin cambiar funciones existentes. Pero si lo que necesitamos es agregar nuevas funciones sobre un conjunto fijo de clases entonces es mejor usar estructuras de datos simples con funciones que operen sobre ellas.

Lección 3: La ley de Demeter es una heurística sencilla que permite mantener nuestros códigos flexibles. Un código es flexible cuando al cambiar una clase *pocas* cosas se rompen.

Resumen

Lección 1: Hay que pensar con cuidado qué dejamos público en nuestras clases. Todo método/atributo público nos quita libertad para modificar nuestras clases a futuro.

Lección 2: Un programador experimentado reconoce que herencia y polimorfismo no son siempre la solución apropiada a un problema. Polymorfismo hace que sea fácil agregar nuevas clases sin cambiar funciones existentes. Pero si lo que necesitamos es agregar nuevas funciones sobre un conjunto fijo de clases entonces es mejor usar estructuras de datos simples con funciones que operen sobre ellas.

Lección 3: La ley de Demeter es una heurística sencilla que permite mantener nuestros códigos flexibles. Un código es flexible cuando al cambiar una clase *pocas* cosas se rompen. Dado que la ley de Demeter limita los métodos que podemos llamar desde una clase también se limita el número de clases que se puedan romper debido a que un método es cambiado.

¿Cómo afecta esto a la E2?

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

- Traten que todo sea privado.

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

- Traten que todo sea privado.

¿Qué aspectos de la E2 deben ser resueltos usando herencia/polimorfismo y qué aspectos deben ser resueltos usando programación basada en procedimientos?

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

- Traten que todo sea privado.

¿Qué aspectos de la E2 deben ser resueltos usando herencia/polimorfismo y qué aspectos deben ser resueltos usando programación basada en procedimientos?

- En cada entrega, hay que agregar nuevas habilidades y nuevas afinidades pero el tamaño del tablero siempre es el mismo.

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

- Traten que todo sea privado.

¿Qué aspectos de la E2 deben ser resueltos usando herencia/polimorfismo y qué aspectos deben ser resueltos usando programación basada en procedimientos?

- En cada entrega, hay que agregar nuevas habilidades y nuevas afinidades pero el tamaño del tablero siempre es el mismo.

¿Cómo elimino los train wrecks de mi programa?

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

- Traten que todo sea privado.

¿Qué aspectos de la E2 deben ser resueltos usando herencia/polimorfismo y qué aspectos deben ser resueltos usando programación basada en procedimientos?

- En cada entrega, hay que agregar nuevas habilidades y nuevas afinidades pero el tamaño del tablero siempre es el mismo.

¿Cómo elimino los train wrecks de mi programa?

- Usa variables auxiliares.

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

- Traten que todo sea privado.

¿Qué aspectos de la E2 deben ser resueltos usando herencia/polimorfismo y qué aspectos deben ser resueltos usando programación basada en procedimientos?

- En cada entrega, hay que agregar nuevas habilidades y nuevas afinidades pero el tamaño del tablero siempre es el mismo.

¿Cómo elimino los train wrecks de mi programa?

- Usa variables auxiliares.

¿Cómo cumplo la ley de Demeter?

¿Cómo afecta esto a la E2?

¿Qué atributos/métodos deben ser privados y cuáles públicos?

- Traten que todo sea privado.

¿Qué aspectos de la E2 deben ser resueltos usando herencia/polimorfismo y qué aspectos deben ser resueltos usando programación basada en procedimientos?

- En cada entrega, hay que agregar nuevas habilidades y nuevas afinidades pero el tamaño del tablero siempre es el mismo.

¿Cómo elimino los train wrecks de mi programa?

- Usa variables auxiliares.

¿Cómo cumplo la ley de Demeter?

- Hay que analizar qué método corresponder agregar a la clase que rompe Demeter.

¿Cuáles son los descuentos típicos en el proyecto debido al cap. 6?

Descuentos más comunes:

- Utilizas un diseño que no respete los principios SOLID
- Crear clases híbridad
- Crear métodos y/o atributos públicos con `access modifiers` incorrectos.
- Violaciones a la ley de Demeter
- Tener `train wrecks`

Actividad

Actividad

El proyecto `Letters` emula una parte chica de un juego similar al del proyecto.

Actividad

El proyecto `Letters` emula una parte chica de un juego similar al del proyecto.

Básicamente, existen cuatro stats: `Atk`, `Def`, `Res` y `Spd`. También hay efectos que cambian sus valores. Cada efecto está asociado a una letra:

- `a`: Aumenta `Atk` en 5.
- `b`: Aumenta `Spd` en 2.
- ...
- `g`: Aumenta `Atk`, `Def`, `Res` y `Spd` en 1.

Actividad

El proyecto `Letters` emula una parte chica de un juego similar al del proyecto.

Básicamente, existen cuatro stats: `Atk`, `Def`, `Res` y `Spd`. También hay efectos que cambian sus valores. Cada efecto está asociado a una letra:

- `a`: Aumenta `Atk` en 5.
- `b`: Aumenta `Spd` en 2.
- ...
- `g`: Aumenta `Atk`, `Def`, `Res` y `Spd` en 1.

Todas las stats parten en cero. El programa recibe un string (e.j., `"bda"`) y ejecuta los efectos de cada caracter en el string (e.j., primero `b`, luego `d` y luego `a`). Finalmente, se retornan los valores de las stats después de aplicados los efectos.

Actividad

El diseño actual se basa en una cadena de ifs:

```
public class Game
{
    private int _atk = 0;
    private int _spd = 0;
    private int _res = 0;
    private int _def = 0;
    private string _actions;

    public Game(string actions)
        => _actions = actions;

    public string Play()
    {
        foreach (var action in _actions)
        {
            if (action == 'a')
                _atk += 5;
            if (action == 'b')
                _spd += 2;
            /* ... */
        }
        return $"{_atk}-{_spd}-{_res}-{_def}";
    }
}
```

Actividad

Lamentablemente, este diseño no escala bien. Es un diseño que nos obliga a modificar nuestro código cada vez que agregamos un nuevo efecto.

Actividad

Lamentablemente, este diseño no escala bien. Es un diseño que nos obliga a modificar nuestro código cada vez que agregamos un nuevo efecto.

De momento es fácil, pero a medida que se agreguen efectos cada vez más complejos este tipo de solución se convertirá en un dolor de cabeza.

Actividad

Lamentablemente, este diseño no escala bien. Es un diseño que nos obliga a modificar nuestro código cada vez que agregamos un nuevo efecto.

De momento es fácil, pero a medida que se agreguen efectos cada vez más complejos este tipo de solución se convertirá en un dolor de cabeza.

Objetivo: Refactoriza el código utilizando herencia. La idea es que llegues a un diseño que respecte el open-closed principle para que, a futuro, sea fácil agregar nuevos efectos sin modificar el funcionamiento del código principal.

Actividad

Lamentablemente, este diseño no escala bien. Es un diseño que nos obliga a modificar nuestro código cada vez que agregamos un nuevo efecto.

De momento es fácil, pero a medida que se agreguen efectos cada vez más complejos este tipo de solución se convertirá en un dolor de cabeza.

Objetivo: Refactoriza el código utilizando herencia. La idea es que llegues a un diseño que respecte el open-closed principle para que, a futuro, sea fácil agregar nuevos efectos sin modificar el funcionamiento del código principal.

... La clave es entender donde usar EDD y donde OOP