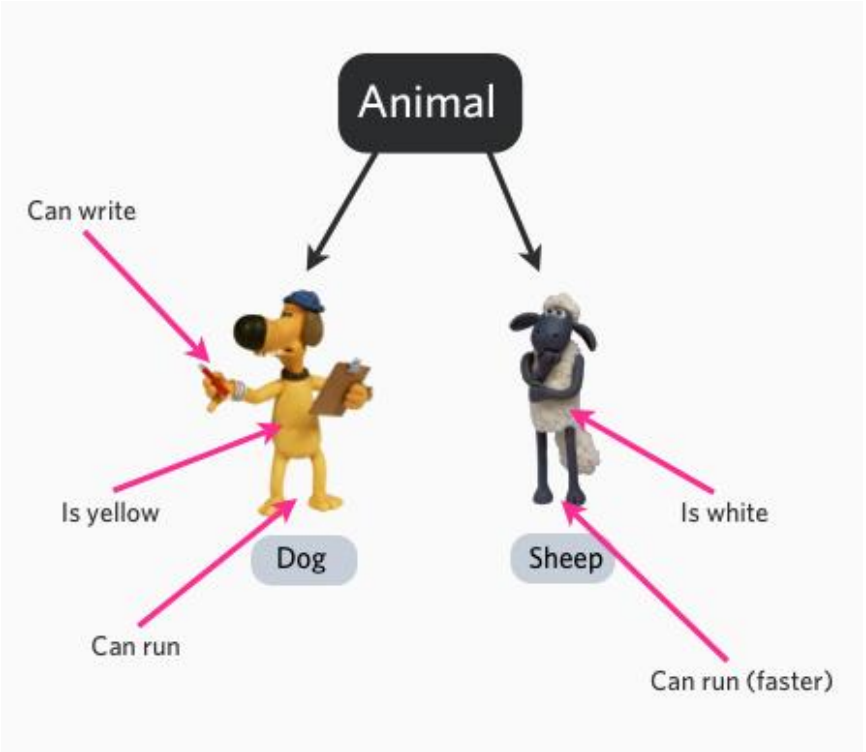
ERICSSON

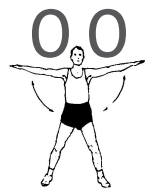
CALISTENIA DE OBJETOS CON C++

USING STD::CPP 2014



anibal.caceres@ericsson.com
@anibal_caceres
ERICSSON R&D

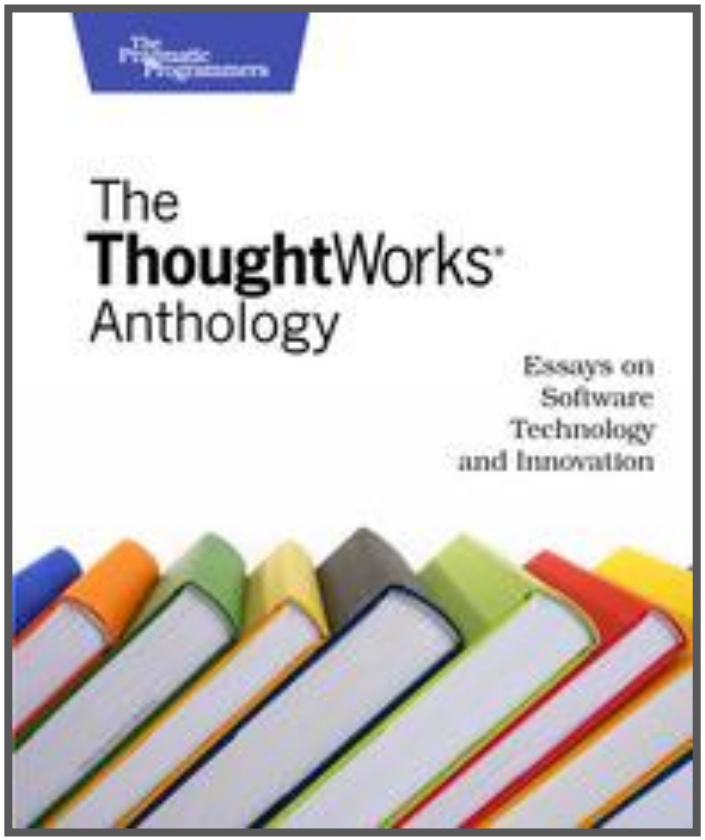
AGENDA



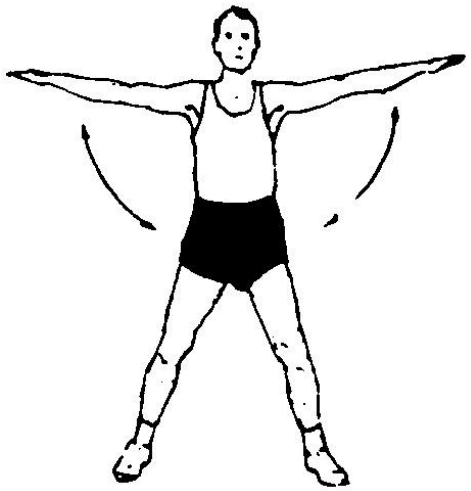
- ¿DE DÓNDE SALE ESTO?
- LA PROMESA
- EL DESAFÍO
- LAS REGLAS
- CONCLUSIÓN
- ¿ALGUNA PREGUNTA?

Puedes preguntar lo que desees durante la presentación…

¿DE DÓNDE SALE ESTO?



"Object Calisthenics", ensayo de Jeff Bay



Calistenia == Ejercicio

LA PROMESA

La programación orientada a objetos nos librará de nuestro antiguo código procedural, permitiéndonos escribir software de manera incremental, y reusable.

EL DESAFÍO

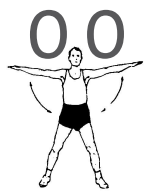
9 reglas estrictas (y extremas).

Escribe unas 1000 líneas de código siguiéndolas...

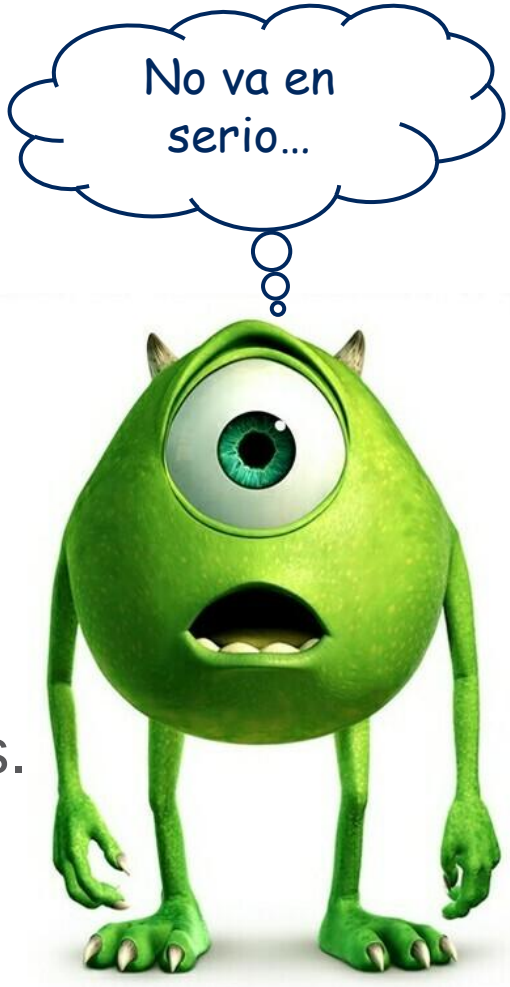
...y saca tus propias conclusiones (verdadero valor del ejercicio),

después usa estas reglas como guías (si te parecen beneficiosas).

LAS 9 REGLAS



1. Un nivel de indentación por función.
2. No uses `else`.
3. Envuelve todas las primitivas y strings.
4. Colecciones de primera clase.
5. Un punto (o flecha) por línea.
6. No abrevies.
7. Mantén todas las entidades pequeñas.
8. Ninguna clase con más de dos variables.
9. No uses “getters” ni “setters”.



REGLA 1: UN NIVEL DE INDENTACIÓN POR FUNCIÓN

Una función gigante suele significar falta de cohesión

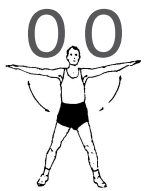
“Single Responsability”

Ventajas al partir: más reusabilidad y legibilidad

REGLA 1: ANTES


```
class Horses {
public:
   // ...

   void shoe() {
      for (auto& x : horsesM) {
         std::cout << "Shoes for " << x.name << ": " << std::endl;
         for (auto i=0; i < 4; i++) {
            std::cout << "Put horseshoe " << (i+1) << std::endl;
         }
      }
      std::cout << "All the horses ready!!" << std::endl;
   }

private:
   std::vector<Horse> horsesM;
}
```

REGLA 1: DESPUÉS



```
class Horses {
public:
   // ...

   void shoe() {
      for (auto& x : horsesM) {
         shoeHorse(x);
      }
      std::cout << "All the horses ready!!" << std::endl;
   }
   void shoeHorse(Horse &horse) {
      std::cout << "Shoes for " << horse.name << ": " << std::endl;
      for (auto i = 0; i < 4; i++) {
         std::cout << "Put horseshoe " << (i+1) << std::endl;
      }
   }

private:
   std::vector<Horse> horsesM;
}
```

REGLA 2: NO USES `else`

- La lógica condicional está “en nuestras venas”
- Duplicidades, repetición
- El Polimorfismo puede ayudar (patrón estrategia)

REGLA 2: ANTES


```
class Animal {
public:
    // ...

   std::string talk() {
      std::string voice;
      if (kindM == “dog")
         voice = “Bark!!";
      else if (kindM == “sheep")
         voice = “Baa!!";
      else if (kindM == "horse")
         voice = "Neigh!!";
      else
         voice = "";
      return voice;
   }

private:
    std::string kindM ;
}
```

REGLA 2: DESPUÉS

```
class Animal {
public:
   virtual ~Animal(){};
   virtual std::string talk() = 0;
}
```

```
class Dog: public Animal {
public:
   Dog(std::string name) :
      nameM(name) {}
   virtual std::string talk() {
      return("Bark!!");
   }
private:
   std::string nameM;
}
```

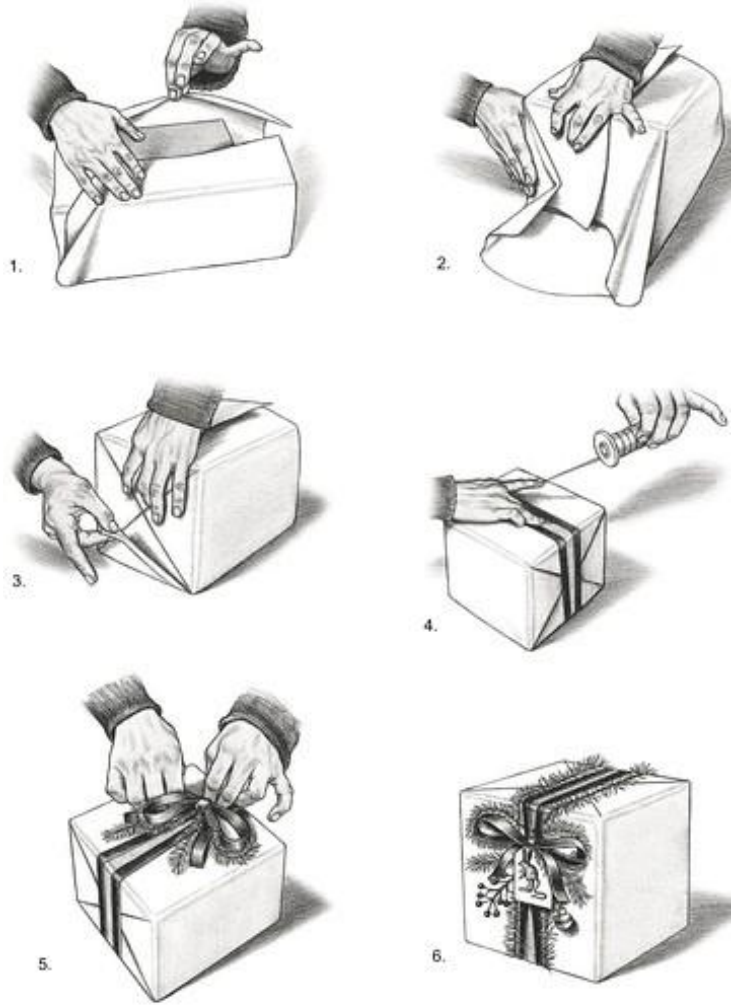
```
class Sheep: public Animal {
public:
   Sheep() {}
   virtual std::string talk() {
      return("Baa!!");
   }
}
```

REGLA 3: ENVUELVE TODAS LAS PRIMITIVAS Y STRINGS

Un `int` o un `string` no tienen semántica asociada

Clase envoltorio: da semántica y atrae comportamientos

Esto equivale al anti-patrón “Primitive Obsession”



REGLA 3: ANTES

```
class Pig: public Animal {
public:
   Pig() {
      weightM = 0;
   }

   virtual std::string talk() {
      return("Oink!!");
   }
   void setWeight(float weight) {
      weightM = theWeight;
   }
   float calculatePrice(float kgPrice) {
      return kgPrice * weightM;
   }

private:
   float weightM;
}
```

REGLA 3: DESPUÉS (I)

```
class Pig: public Animal {
public:
   Pig() {
      weightM = Weight();
   }

   virtual std::string talk() {
      return("Oink!!");
   }
   void setWeight(Weight& weight) {
      weightM = theWeight;
   }
   Weight getWeight() {
      return weightM;
   }
   Money calculatePrice(Money& kgPrice) {
      return kgPrice * weightM.valueAs(Weight::KG);
   }
private:
   Weight weightM;
}
```

REGLA 3: DESPUÉS (II)

```
class Weight {
public:
   enum Unit { KG, LB };

   Weight(float amount, Unit unit = Weight::KG) :
      amountM(amount), unitM(unit) {}
   virtual ~Weight() {}

   float valueAs(Unit theUnit) {
      return amountM * conversionM[unitM][theUnit];
   }
   bool operator>(Weight& weight) {
      return amountM > weight.valueAs(unitM);
   }

private:                                    //      KG    LB
   static const float conversionM[2][2] = { 1.00, 2.20,   // KG
                                            0.45, 1.00 };// LB
   float amountM;
   Unit unitM;
}
```

REGLA 3: DESPUÉS (III)

```
class Money {
public:
   enum Currency { EUR, GBP, USD };

   Money(float amount, Currency currency = Money::EUR) :
      amountM(amount), currencyM(currency) {}
   virtual ~Money() {}

   float valueAs(Currency currency) {
      // …Do something fancy, like contact exchange service…
   }
   Money operator*(float multiplier) {
      return Money(amountM*multiplier, currencyM);
   }

private:
   float amountM;
   Currency currencyM;
}
```

REGLA 4: COLECCIONES DE PRIMERA CLASE

Cada colección en su propia clase, sin más miembros

Filtros, uniones, ... acabarán en la clase de la colección

Puede cambiarse la colección sin afectar a sus usuarios

REGLA 4: ANTES


```
class Farmer {
public:
   // ...

   void addPig(Pig& pig) {
      pigsM.push_back(pig);
   }
   std::vector<Pig> findPigsToSell() {
      std::vector<Pig> pigsToSell;
      Weight min(200);
      for (auto& x : pigsM) {
         if (x.getWeight() > min)
            pigsToSell.push_back(x);
      }
      return pigsToSell;
   }
private:
   std::vector<Pig> pigsM;
   // ...
}
```

REGLA 4: DESPUÉS

```
class Farmer {
public:
   // ...

   void addPig(Pig& pig) {
      pigsM.add(pig);
   }
   Pigs findPigsToSell()  {
      Weight min(200);
      return pigsM.biggerThan(min);
   }

private:
   Pigs pigsM;
}
```

```
class Pigs {
public:
   Pigs() {}
   virtual ~Pigs() {}

   void add(Pig &aPig) {
      pigsM.push_back(aPig);
   }
   Pigs biggerThan(Weight& min) {
      Pigs toRet;
      for (auto& x : pigsM) {
         if (x.getWeight() > min)
            toRet.add(x);
      }
      return toRet;
   }

private:
   std::vector<Pig> pigsM;
};
```

REGLA 5: UN PUNTO (O FLECHA) POR LÍNEA

Muchos puntos por línea ≡ responsabilidad desplazada

Objetos que miran dentro de otros: saben demasiado

Ley de Deméter: “habla sólo con tus amigos”

REGLA 5

› Antes:

```
Dog dog("Toby");
// Let's sit down the dog.
dog.getLegs().backLegs().down();
```

› Después:

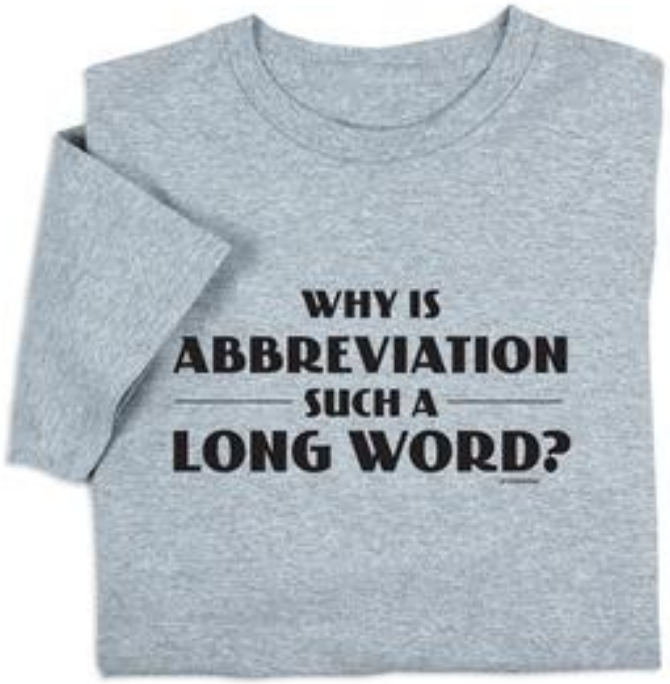
```
Dog dog("Toby");
// Let's sit down the dog.
dog.sitDown();
```

REGLA 6: NO ABREVIES

Abreviar dificulta la legibilidad, y puede ocultar problemas

Suele indicar duplicidad, o responsabilidad desplazada

En `Order` no crees un `shipOrder()`: mejor `ship()`



REGLA 7: MANTÉN TODAS LAS ENTIDADES PEQUEÑAS

- No más de 50 líneas por clase (definición)
- Agrupar esas clases pequeñas en namespaces
- No más de 10 clases por namespace

REGLA 8: NINGUNA CLASE CON MÁS DE 2 VARIABLES

Clases que encapsulan 1 variable o clases que coordinan 2 variables

Las clases con más de 2 variables miembro tienen menos cohesión

Modelo de objetos más eficaz con varios objetos colaboradores

REGLA 8

› Antes:

```
class Name {
  // ...
  std::string first;
  std::string middle;
  std::string last;
}
```

› Después:

```
class Name {
  // ...
  FamilyName family;
  GivenName given;
}
```

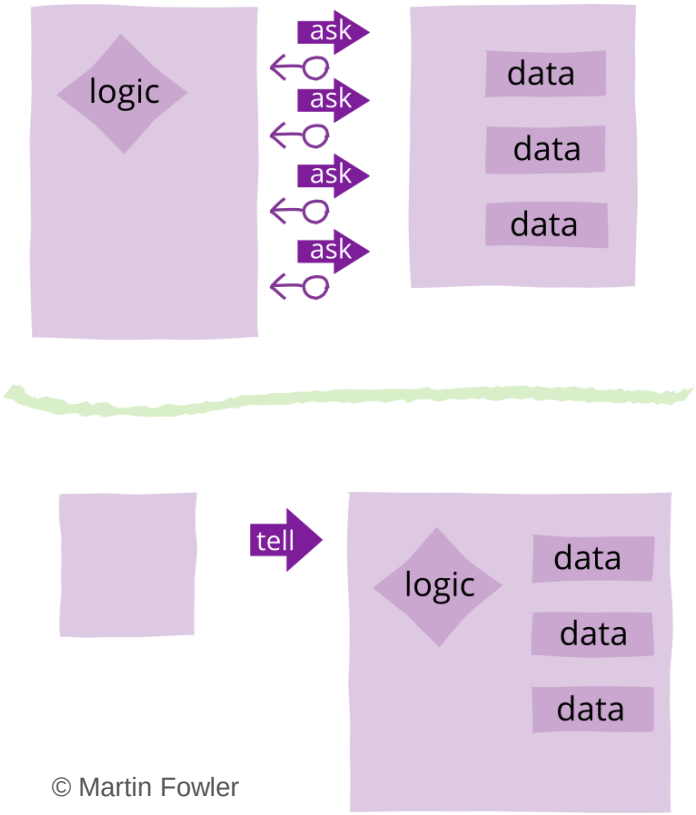
```
class FamilyName {
  // ...
  std::string family;
};

class GivenName {
  // ...
  std::vector<std::string> given;
}
```

REGLA 9: NO USES "GETTERS" NI "SETTERS"

Encapsulación extrema

Usando `get()` alejamos datos y comportamientos

"Tell, don't ask"



© Martin Fowler

REGLA 9: ANTES


```
class Cow: public Animal {
public:
   Cow() {
      milkM = Volume(0);
   }
   virtual ~Cow();

   std::string talk() {
      return “Moo!!";
   }
   Volume getMilk() {
      return milkM;
   }
   void setMilk(Volume& milk) {
      milkM = milk;
   }
private:
   Volume milkM;
};
```

```
Cow myCow = Cow();
Volume milkToday = Volume(20);
Volume milkTotal =
   myCow.getMilk() + milkToday;
myCow.setMilk(totalMilk);
```

REGLA 9: DESPUÉS


```
class Cow: public Animal {
public:
   Cow() {
      milkM = Volume(0);
   }
   virtual ~Cow();

   std::string talk() {
      return “Moo!!";
   }
   void addMilk(Volume& milk) {
      milkM  = milkM + milk;
   }

private:
   Volume milkM;
};
```

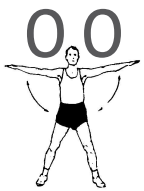
```
Cow myCow = Cow();
Volume milkToday = Volume(20);
myCow.addMilk(milkToday);
```

CONCLUSIÓN

- 7 reglas sobre encapsulación, 1 sobre polimorfismo, y 1 sobre nombrado adecuado.
- Objetivo buscado: cohesión, DRY, legibilidad, mantenibilidad.
- Aplicar y analizar, no forzar si no estamos cómodos.
- No ser extremista!!
- Principal problema: posible "estallido" de clases y funciones.

Ponle AMOR a tu CÓDIGO: Quiere lo que Escribes

¿ALGUNA PREGUNTA?



¡¡GRACIAS por vuestra atención!!

¿Algún comentario más?, por favor no dudéis en contactar conmigo ^_^