PROGRAMACIÓN CON OBJETOS

¿Cuál es la ventaja principal de la **Orientación a Objetos**?

Ocultar la implementación

¿Cómo consigue la OO ocultar la implementación?

Principios básicos de la construcción de software (OO)

- **Abstracción**: diferenciar el *qué* y el *cómo*
- **Modularidad**: componentes, módulos (en OO, clases y objetos), interfaces, etc.
- △ **cohesión**: módulos auto-contenidos, independientes y con un único propósito
- ▽ **acoplamiento**: reducir las dependencias entre módulos

Versión inicial: Lista v0.1

En relación a los principios de *alta cohesión* y *bajo acoplamiento*, criticar la implementación siguiente en Java de un TAD Lista:

```
public abstract class List<T> {
  public void addFirst(T value) { ... };
  public void removeFirst() { ... };
  public void addLast(T value) { ... };
  public void removeLast() { ... };
  public T first() { ... };
  public T last() { ... };
  public boolean isEmpty() { ... };
  public int length() { ... };
  public List<T> clone() { ... };
  public boolean isEqualTo(List<T>) { ... };
  public abstract void traverse();
  // etc...
}
```

Abstracción

- La clase abstracta `List<T>` diferencia entre el *qué* y el *cómo*
- Qué hace la lista vs. cómo se almacenan los elementos

Cohesión

> Cohesion refers to the degree to which the elements inside a module belong together
>
> --- E. Yourdon & L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, 2nd edition, 1986.

Discusión sobre la implementación

v0.1 • críticas → v0.2 • críticas → v0.3 • críticas → v0.4 → resumen

Críticas a Lista v0.1

- `List<T>` aglutina más de una responsabilidad: almacenar y recorrer. La implementación no parece cohesionada.
- El método `traverse()` proporciona una interfaz a los métodos que implementen el recorrido de la lista ¿para hacer qué?
- ¿Y si hay distintas implementaciones de `traverse()`? Si implementamos varias versiones de la lista, introducimos más dependencias (acoplamiento)

Problemáticas de Lista v0.1

- **Cohesión** baja
- **Variabilidad** no bien tratada $\longrightarrow$ poca **flexibilidad**

Implementación alternativa: Lista v0.2

Hay que crear nuevos tipos de recorrido. Ampliamos la interfaz...

```
public interface List<T> {
  public void addFirst(T value);
  public void removeFirst();
  public void addLast(T value);
  public void removeLast();
  public T first();
  public T last();
  public boolean isEmpty();
  public int length();
  public List<T> clone();
  public boolean isEqualTo(List<T>);
  ...
```

```
  ...
  public void traverseForward();
  public void traverseBackWard();
  public void traverseEvens(); //pares
  public void traverseOdds();  //impares
}
```

Críticas a Lista v0.2

- Si hay que cambiar la operación básica que hace `traverse()` con cada elemento (imprimir, sumar, etc.), ¿cuántos métodos hay que cambiar? Hay muchas dependencias
- Cuanto más variedad de recorridos (la interfaz es mayor), menos flexibilidad para los cambios. Implementación poco flexible

Problemáticas de Lista v0.2

- **Acoplamiento** excesivo: muchas **dependencias** (provocadas por el exceso de herencia)
- **Flexibilidad** escasa

Implementación alternativa: Lista v0.3

Delegar funcionalidad hacia las subclases (vía **herencia**).

Criticar la implementación:

```
class ListForward<T> extends List<T> {
  //...
  public void traverse() { // recorrer hacia adelante };
}
class ListBackward<T> extends List<T> {
  //...
  public void traverse() { // recorrer hacia atras};
}
```

Críticas a Lista v0.3

- ¿Hay que especializar de nuevo para cada tipo de operación que hace `traverse()` con cada elemento individual (imprimir, sumar, etc.)?
- ¿Y si hay que especializar de nuevo el recorrido: sólo los pares, sólo los impares, etc.?

Problemáticas de Lista v0.3

- **Acoplamiento** elevado: Si hay que crear nuevos tipos de recorrido, se abusará de la herencia para crear *estructuras* complejas
- **Variabilidad** no está bien tratada: poca **flexibilidad**, baja **reutilización**

¿Cómo se resuelve esto en las bibliotecas típicas que conocéis (v.g. C++ STL, Java Collections, etc.)?

Implementación alternativa: Lista v0.4

Delegar hacia otra clase

```
public interface List<T> {
  void addFirst(T value);
  void removeFirst();
  void addLast(T value);
  void removeLast();
  T first();
  T last();
  boolean isEmpty();
  int length();
  List<T> clone();
  boolean isEqualTo(List<T>);
  Iterator<T> iterator();
}
```

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

Ventajas de Lista v0.4

- Mayor **cohesión**: Las responsabilidades están ahora separadas: `List` almacena, `Iterator` recorre. `List` está más cohesionada
- Para hacer `List` más cohesionada, se ha tenido que introducir una **dependencia** (acoplamiento)
- Uso de **delegación** (o *composición*) en lugar de la herencia: la responsabilidad de recorrer se ha delegado hacia otro sitio

Resumen de problemas

Problema	desde v0.1 ...	... hasta v0.4
Cohesión	Baja: `List<T>` aglutina almacenamiento y recorrido	Alta: `List<T>` solo almacena; `Iterator<T>` recorre
Variabilidad	No tratada: difícil cambiar la forma de recorrer	Fácil crear nuevos `Iterator` sin tocar `List`
Flexibilidad	Poca: cambios en recorrido afectan a `List`	Mayor: Cambios aislados en `Iterator`
Acoplamiento	Alto: `List` depende de cómo se recorre	Bajo: separación clara de responsabilidades

Ocultar la implementación

Los principios aplicados han sido:

- **Abstracción**: diferenciar el *qué* y el *cómo*
- **Cohesión** (maximizar): módulos auto-contenidos, independientes y con un único propósito
- **Acoplamiento** (minimizar): dependencias entre módulos
- **Modularidad**: clases, interfaces y componentes/módulos

Alta cohesión, bajo acoplamiento

> Cuando los componentes están aislados, puedes cambiar uno sin preocuparte por el resto. Mientras no cambies las interfaces externas, no habrá problemas en el resto del sistema
>
> --- E. Yourdon & L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall, 2nd edition, 1986.

Modularidad

Reducir el acoplamiento usando módulos o componentes con distintas responsabilidades, agrupados en bibliotecas

Técnicas de ocultación

- **Encapsular**: agrupar en módulos y clases
- **Visibilidad**: `public`, `private`, `protected`, etc.
- **Delegación**: incrementar la cohesión extrayendo funcionalidad pensada para otros propósitos fuera de un módulo
- **Herencia**: delegar *en vertical*
- **Polimorfismo**: ocultar la implementación de un método, manteniendo la misma interfaz de la clase base
- **Interfaces**: usar interfaces bien documentadas

Herencia

- **Reutilizar la interfaz**
  - Clase base y derivada son del mismo tipo
  - Todas las operaciones de la clase base están también disponibles en la derivada
- **Redefinir vs. reutilizar el comportamiento**
  - *Overriding* (redefinición o sobreescritura): cambio de comportamiento
  - *Overloading* (sobrecarga): cambio de interfaz
- **Herencia pura vs. extensión**
  - Herencia pura: mantiene la interfaz tal cual (relación *es-un*)
  - Extensión: amplía la interfaz con nuevas funcionalidades(relación *es-como-un*). Puede causar problemas de *casting*.

Generalización y especialización

> When you inherit, you take an existing class and make a special version of it. In general, this means that you're taking a general-purpose class and specializing it for a particular need. [...] it would make no sense to compose a car using a vehicle object —a car doesn't contain a vehicle, it is a vehicle. The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.
>
> --- Bruce Eckel

Polimorfismo

Fenómeno por el que, cuando se llama a una operación de un objeto del que no se sabe su tipo específico, se ejecuta el método adecuado de acuerdo con su tipo.

El polimorfismo se basa en:

- **Enlace dinámico** (*dynamic binding*): se elige el método a ejecutar en tiempo de ejecución, en función de la clase de objeto; es la implementación del *polimorfismo*

Overriding

- En general, en un lenguaje OO es posible sobreescribir o redefinir (*override*) los métodos heredados de una superclase.
- En algunos lenguajes es obligatorio (en otros es recomendado) especificar explícitamente cuándo un método es redefinido.
- Vamos a ver ejemplos en distintos lenguajes...

Ejemplo 1: Override en Scala

```
class Complejo(real: Double, imaginaria: Double) {
  def re = real
  def im = imaginaria
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

- Cuando se redefine un método abstracto, no es necesario `override`
- Pero si se quiere redefinir un método concreto, `override` es necesario para evitar sobreescrituras accidentales.
- En Scala, el riesgo de redefinición accidental de métodos es mayor debido a los mixins (`trait` en Scala).

Scala Traits

Un **trait** es una forma de separar las dos principales responsabilidades de una clase: definir el **estado** de sus instancias y definir su **comportamiento**.

- Las clases y los objetos en Scala pueden extender un `trait`
- Los `trait` de Scala son similares a las `interface` de Java.
- Los `trait` no pueden instanciarse
- Los métodos definidos en una clase tienen precedencia sobre los de un `trait`
- Los `trait` no tienen estado propio, sino el del objeto o la instancia de la clase a la que se aplica

Ejemplo 2: Un iterador con Scala traits

```
trait Iterator[A] {
  def hasNext: Boolean
  def next(): A
}

class IntIterator(to: Int) extends Iterator[Int] {
  private var current = 0
  override def hasNext: Boolean = current < to
  override def next(): Int =  {
    if (hasNext) {
      val t = current
      current += 1
      t
    } else 0
  }
}

val iterator = new IntIterator(10)
println(iterator.next())  // prints 0
println(iterator.next())  // prints 1
```

¿Y en Java no hay *traits*?

Java default methods

- Desde Java 8, las interfaces pueden incorporar métodos por defecto que hacen que las interfaces de Java se comporten más como un trait.
- Sirven para implementar herencia múltiple

Ejemplo 3: `@Override` en Java

Este ejemplo en Java es realmente la implementación de un **diseño incorrecto**, pues hay una doble dependencia entre las clases `Real` y `Complejo`.
La frontera entre Diseño e Implementación queda aquí un poco difusa.

```
class Real {
  double re;
  public Real(double real) {
      re = real;
  }
  public double getRe() { return re; }
  /* Probar a comentar el siguiente método y mantener el
     Override de Complejo::sum(Real other) */
  public Real sum(Real other) {
    return new Real(re + other.getRe());
  }
  /* Probar a comentar el siguiente método y mantener el
     Override de Complejo::sum(Complejo other) */
  public Complejo sum(Complejo other) {
    return new Complejo( re + other.getRe(), other.getIm() );
  }
  public String toString() {
    return String.format("%.1f", re);
  }
}
```

```
class Complejo extends Real {
  double im;
  public Complejo(double real, double imaginaria) {
    super(real);
    im = imaginaria;
  }
  @Override
  public Complejo sum(Real other) {
    return new Complejo( re + other.getRe(), im );
  }
  @Override
  public Complejo sum(Complejo other) {
    return new Complejo( re + other.getRe(), im + other.getIm() );
  }
  public Double getIm() { return im; }
  public String toString() {
    return String.format("%.1f", re) + ((im < 0)? "" : "+") +
       String.format("%.1f", im) + "i";
  }
}
```

```
public class Main {
  public static void main(String args[]) {
      Real r = new Real(7.6);
      Complejo c = new Complejo(1.2, 3.4);
      System.out.println("Número real: " + r);
      System.out.println("Número complejo: " + c);
      System.out.println("Número complejo: " + c.sum(r) );
      System.out.println("Número complejo: " + r.sum(c) );
  }
}
```

¿Qué sucede si no ponemos `@Override` a los métodos redefinidos?

Si no se añade `@Override`, podemos confundirnos y hacer un *overload* accidental de un método cuando realmente queríamos redefinirlo.

Ejemplo 4: Override en C#

- `DescribeCar` muestra una descripción básica de un coche y llama a `ShowDetails` para información adicional.
- Cada clase define su propia versión de `ShowDetails`
- Usamos mofificadores `new` y `override` distintos en cada clase `ConvertibleCar` y `Minivan`

```
class Car
{
  public void DescribeCar()
  {
    System.Console.WriteLine("Four wheels and an engine.");
    ShowDetails();
  }

  public virtual void ShowDetails()
  {
    System.Console.WriteLine("Standard transportation.");
  }
}
```

```
class ConvertibleCar : Car
{
  public new void ShowDetails()
  {
    System.Console.WriteLine("A roof that opens up.");
  }
}

class Minivan : Car
{
  public override void ShowDetails()
  {
    System.Console.WriteLine("Carries seven people.");
  }
}
```

```
public static void TestCars1()
{
  System.Console.WriteLine("\nTestCars1\n----------");

  var cars = new List<Car> {
        new Car(),
        new ConvertibleCar(),
        new Minivan() };

  foreach (var car in cars)
  {
    car.DescribeCar();
    System.Console.WriteLine("----------");
  }
}
```

`TestCars` produce la salida siguiente:

```
TestCars1
----------
Four wheels and an engine.
Standard transportation.
----------
Four wheels and an engine.
Standard transportation.
----------
Four wheels and an engine.
Carries seven people.
----------
```

¿Los resultados son los esperados?

- El tipo del segundo objeto de la lista es `ConvertibleCar`, pero `DescribeCar` no accede a la versión de `ShowDetails` definida en `ConvertibleCar` (debido a `new`).
- El tipo del tercer objeto de la lista es `Minivan`, que redefine con `override` el método `ShowDetails` declarado en la clase base.

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2\n----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3\n----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

Estos métodos producirían las salidas siguientes:

```
TestCars2
----------
A roof that opens up.
Carries seven people.

TestCars3
----------
Standard transportation.
Carries seven people.
```

- En `TextCars2`, el tipo de los objetos creados coincide con el tipo declarado.
- En `TextCars3`, el tipo de los objetos creados es una subclase de la clase del tipo declarado.

¿Sobre qué mecanismo funciona el polimorfismo?

Tipado

- Tipado estático vs. dinámico
  - Tipado estático: el tipo de cada variable se conoce en tiempo de compilación
  - Tipado dinámico (*duck typing*): el tipo de cada variable se conoce en tiempo de ejecución
- Contrato nominal vs. estructural
  - Contrato nominal: hay que escribir un tipo explícito
  - Contrato estructural: el tipo se deduce de la estructura de métodos

Moldes o *casting* de tipos

- *Upcasting:* Interpretar un objeto de una clase derivada como del mismo tipo que la clase base
- *Downcasting:* Interpretar un objeto de una clase base como del mismo tipo que una clase derivada suya

Ejemplo de casting: Aventura v0.1

```
public class PersonajeDeAccion {
  public void luchar() {}
}

public class Heroe extends PersonajeDeAccion {
  public void luchar() {}
  public void volar() {}
}

public class Creador {
  PersonajeDeAccion[] personajes() {
    PersonajeDeAccion[] x = {
      new PersonajeDeAccion(),
      new PersonajeDeAccion(),
      new Heroe(),
      new PersonajeDeAccion()
    };
    return x;
  }
}
```

```
public class Aventura {
  public static void main(String[] args) {
    PersonajeDeAccion[] cuatroFantasticos = new Creador().personajes();
    cuatroFantasticos[1].luchar();
    cuatroFantasticos[2].luchar(); // Upcast

    // En tiempo de compilacion: metodo no encontrado:
    //! cuatroFantasticos[2].volar();
    ((Heroe)cuatroFantasticos[2]).volar(); // Downcast
    ((Heroe)cuatroFantasticos[1]).volar(); // ClassCastException
    for (PersonajeDeAccion p: cuatroFantasticos)
        p.luchar; // Sin problema
    for (PersonajeDeAccion p: cuatroFantasticos)
        p.volar; // El 0, 1 y 3 van a lanzar ClassCastException
  }
}
```

Críticas a Aventura v0.1

- ¿De qué tipos van a ser los personales de acción? $\rightarrow$ problema de *downcasting*
- Hay que rediseñar la solución por ser insegura

Ejemplo de casting: Aventura v0.2

```
interface SabeLuchar {
  void luchar();
}
interface SabeNadar {
  void nadar();
}
interface SabeVolar {
  void volar();
}
class PersonajeDeAccion {
  public void luchar() {}
}
class Heroe
    extends PersonajeDeAccion
    implements SabeLuchar,
               SabeNadar,
               SabeVolar {
  public void nadar() {}
  public void volar() {}
}
```

```
public class Aventura {
  static void t(SabeLuchar x)
    { x.luchar(); }
  static void u(SabeNadar x)
    { x.nadar(); }
  static void v(SabeVolar x)
    { x.volar(); }
  static void w(PersonajeDeAccion x)
    { x.luchar(); }
  public static void main(String[] args)
  {
    Heroe i = new Heroe();
    t(i); // Tratar como un SabeLuchar
    u(i); // Tratar como un SabeNadar
    v(i); // Tratar como un SabeVolar
    w(i); // Tratar como un PersonajeDeAccion
  }
}
```

Aventura v0.3 en C++

```cpp
template <typename T>
concept SabeLuchar = requires(T t) { t.luchar(); };

template <typename T>
concept SabeNadar = requires(T t) { t.nadar(); };

template <typename T>
concept SabeVolar = requires(T t) { t.volar(); };

// Clase base normal
struct PersonajeDeAccion {
    void luchar() { std::cout << "Pum! (Luchando)\n"; }
};

struct Heroe : public PersonajeDeAccion {
    void nadar() { std::cout << "Splash! (Nadando)\n"; }
    void volar() { std::cout << "Whoosh! (Volando)\n"; }
};
```

- En lugar de `interface` (Java), definimos qué requiere el tipo
- `Heroe` hereda código de `PersonajeDeAccion`, pero NO necesita declarar `implements SabeNadar, SabeVolar`
- `Heroe` cumple los concepts automáticamente por tener los métodos

```
// Usamos 'auto&' para pasar por
// referencia (evitar copias).

void t(SabeLuchar auto& x) {
    x.luchar();
}

void u(SabeNadar auto& x) {
    x.nadar();
}

void v(SabeVolar auto& x) {
    x.volar();
}

void w(PersonajeDeAccion& x) {
    x.luchar();
}
```

- `t`, `u`, `v` usan concepts (tipado estructural): aceptan cualquier tipo T que satisfaga el concept
- `w` usa herencia clásica con tipado nominal: acepta solo `PersonajeDeAccion` o sus hijos explícitos (como en Java).

```
int main() {
    Heroe i;

    t(i); // Heroe hereda luchar(), así que cumple SabeLuchar
    u(i); // Heroe tiene nadar(), cumple SabeNadar
    v(i); // Heroe tiene volar(), cumple SabeVolar
    w(i); // Heroe es hijo de PersonajeDeAccion

    return 0;
}
```

- Con `concept` se resuelve en tiempo de compilación, sin necesidad de casting

Uso de la herencia

- Herencia de **interfaz** vs. herencia de **comportamiento** o **implementación**:
  - En Java, `implements` = herencia de interfaz y `extends` = herencia de interfaz + implementación
  - ¿Hay herencia sólo de comportamiento? Pista: pensar en C++
- Herencia como **tipo** vs herencia como **estructura**:
  - En herencia de tipos, cada subclase es un subtipo. Debe satisfacerse el principio de **sustitución** de Liskov: toda operación que funciona para un objeto de la clase C también debe funcionar para un objeto de una subclase de C (subtipado de comportamiento *fuerte*)
  - Usar la herencia como una forma de estructurar programas es **erróneo**, pues provoca que no se satisfaga la propiedad LSP.

¿El polimorfismo está ligado siempre a la herencia?

Polimorfismo paramétrico

- Tipos **genéricos**
  - Ada
  - C++ generics
  - Java *templates* (desde JDK 1.5)
  - Scala
  - etc.
- Diferencias entre lenguajes:
  - En C++, los genéricos permiten meta-programación en tiempo de compilación
  - En Java, las plantillas son wrappers que *moldean* objetos (*syntactic sugar*)

Usos incorrectos de la herencia

Mal ejemplo 1 (Java): herencia como estructura

```
class Account {
  float balance;
  float getBalance() { return balance; }
  void transferIn (float amount) { balance -= amount; }
}

class VerboseAccount extends Account {
  void verboseTransferIn (float amount) {
    super.transferIn(amount);
    System.out.println("Balance: "+balance);
  };
}

class AccountWithFee extends VerboseAccount {
  float fee = 1;
  void transferIn (float amount) { super.verboseTransferIn(amount-fee); }
}
```

- Todos los objetos a de la clase `Account` deben cumplir que si $b = a.getBalance()$ antes de ejecutar $a.transferIn(s)$ y $b' = a.getBalance()$ después de ejecutar $a.transferIn(s)$, entonces $b + s = b'$.
- Sin embargo, con la estructura `AccountWithFee` < `VerboseAccount` < `Account`, un objeto de tipo `AccountWithFee` no funciona bien cuando se contempla como un objeto `Account`. Considérese la siguiente secuencia:

```
void f(Account a) {
  float before = a.getBalance();
  a.transferIn(10);
  float after = a.getBalance();
  // Suppose a is of type AccountWithFee:
  //   before + 10 != after    !!
  //   before + 10-1 = after
}
```

Mal Ejemplo 2 (Scala): herencia de implementación

```scala
abstract class Writer {
  def print(str: String): Unit
}

class ConsoleWriter extends Writer {
  override def print(str: String) = println(str)
}

class UppercaseWriter extends ConsoleWriter {
  override def print(str: String) =
    super.print(str.toUpperCase())
}

object Test {
   def main(args: Array[String]) {
        val writer = new UppercaseWriter
        writer.print("abc")
   }
}
```

Ahora hay que añadir un nuevo comportamiento:

```
class WithSpacesWriter extends ConsoleWriter {
  override def print(str: String) =
    super.print(str.split("").mkString(" "))
}
```

```
object Test {
   def main(args: Array[String]) {
        val writer1 = new UppercaseWriter
        writer1.print("abc")

        val writer2 = new WithSpacesWriter
        writer2.print("abc")
   }
}
```

¿Y si queremos combinar ambas formas de imprimir?

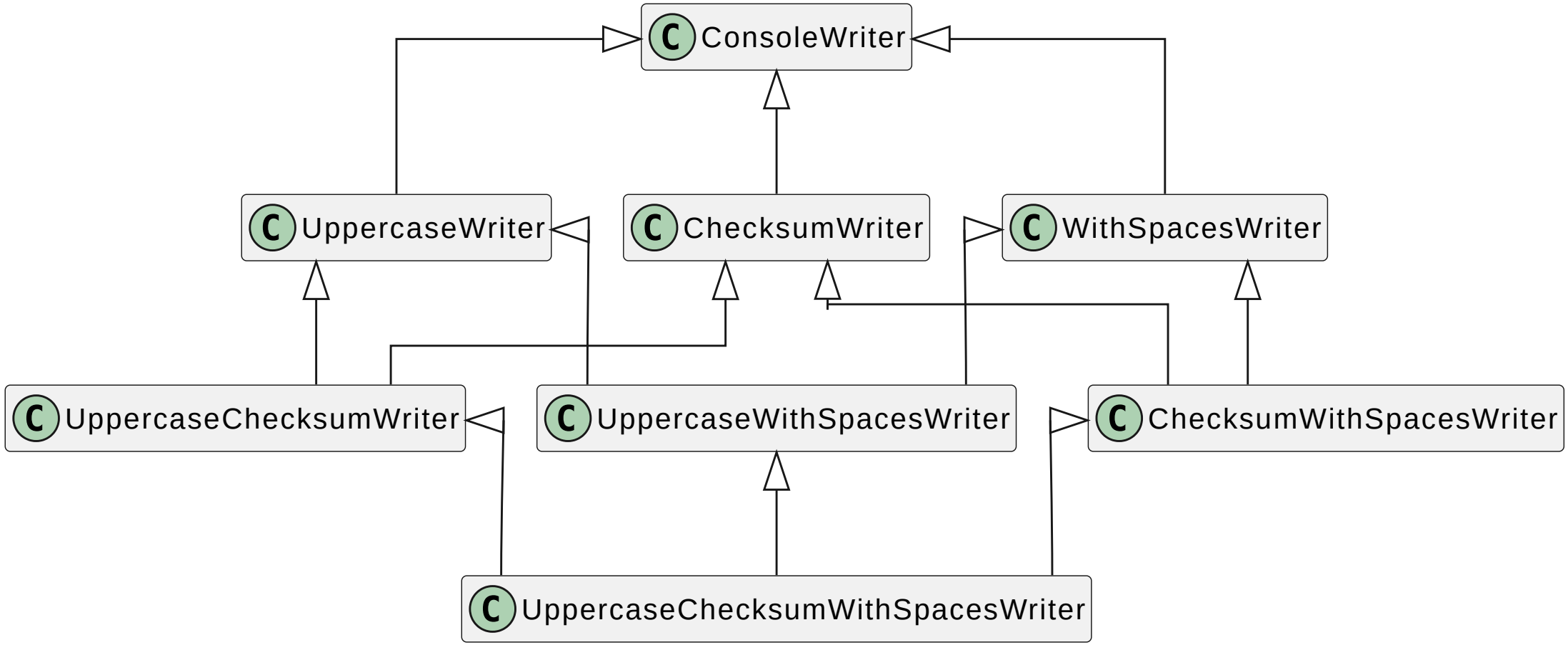
```
class UppercaseWithSpacesWriter extends UppercaseWriter {
  override def print(str: String) =
    super.print(str.split("").mkString(" "))
}

class WithSpacesUppercaseWriter extends WithSpacesWriter {
  override def print(str: String) =
    super.print(str.toUpperCase())
}
```

```
object Test {
   def main(args: Array[String]) {
        val writer1 = new UppercaseWriter
        writer1.print("abc")

        val writer2 = new WithSpacesWriter
        writer2.print("abc")

        val writer3 = new WithSpacesUppercaseWriter
        writer3.print("abc")

        val writer4 = new UppercaseWithSpacesWriter
        writer4.print("abc")
   }
}
```

¿Y si aparece una nueva forma de imprimir?

```
class ChecksumWriter extends ConsoleWriter {
  override def print(str: String) = {
    super.print(MessageDigest.getInstance("MD5").
                digest(str.getBytes("UTF-8")).
                map("%02x".format(_)).
                mkString("","","] ") +
                str )
  }
}
```

Ejemplo: Herencia fuera de control



¡Mal uso de la herencia!

Ejemplo 2 (Scala): herencia de interfaz (traits)

```scala
abstract class Writer {
  def print(str: String): Unit
}

class ConsoleWriter extends Writer {
  override def print(str: String) = println(str)
}

trait Uppercase extends Writer {
  abstract override def print(str: String) =
    super.print(str.toUpperCase())
}

trait WithSpaces extends Writer {
  abstract override def print(str: String) =
    super.print(str.split("").mkString(" "))
}
```

```
object Test {
   def main(args: Array[String]) {
        val writer1 = new ConsoleWriter with Uppercase
        writer1.print("abc")
        val writer2 = new ConsoleWriter with Uppercase with WithSpaces
        writer2.print("abc")
   }
}
```

Genera la salida:

```
ABC
A B C
```

Stackable traits

- En Scala, los `trait` normales son como interfaces, se enlazan en tiempo de ejecución (no tienen acceso a `super`).
- Se pueden redefinir *stackable traits* con `abstract override` para dar acceso a `super`
- `abstract` no es necesario si se redefine un método no abstracto, que ya tiene una implementación
- En diseño, son una implementación del patrón *Decorator* pero por composición de clases en vez de por composición de objetos

Implementación y diseño

Ejemplo 3 (C#): rectángulos versión 0.1

Geométricamente, un cuadrado es un rectángulo, así que usamos herencia pura (*es-un*):

```
public class Rectangle {
  private Point topLeft;
  private double width;
  private double height;

  public double Width {
    get { return width; }
    set { width = value; }
  }

  public double Height {
    get { return height; }
    set { height = value; }
  }
}
```

```
public class Square: Rectangle {
    ...
}
```

Problema: cuadrados como rectángulos

- Matemáticamente, un cuadrado puede **ser-un** rectángulo.
- Pero en Informática un objeto `Square` **no es-un** objeto `Rectangle`
  - Un `Square` no tiene propiedades `height` y `width`.
- Pero supongamos que no nos importa el desperdicio de memoria.
  - `Square` heredará los métodos accesores de `Rectangle`.
- Así que hacemos lo siguiente...

Ejemplo: rectángulos versión 0.2

```
public class Square: Rectangle {
  public new double Width
  {
    set {
      base.Width = value;
      base.Height = value;
    }
  }
  public new double Height
  {
    set {
      base.Height = value;
      base.Width = value;
    }
  }
}
```

Nota: Diferencia entre `new` y `override` en C#

- El comportamiento de un objeto `Square` no es consistente con el de un objeto `Rectangle`:

```
Square s = new Square();
s.SetWidth(1);   // fija ambos
s.SetHeight(2);  // fija ambos

void f(Rectangle r)
{
  r.SetWidth(32); // llama a Rectangle.SetWidth
}
```

- ¿Qué sucede si pasamos un `Square` a la función `f`?

  ¡No cambia `Height`!

- Podría argumentarse que el error era que los métodos `Width` y `Height` no se declararon `virtual` en `Rectangle`.

Ejemplo: rectángulos versión 0.3

Hacemos que los métodos `Width` y `Height` sean `virtual` en C#:

```
public class Rectangle
{
  private Point topLeft;
  private double width;
  private double height;
  public virtual double Width
  {
    get { return width; }
    set { width = value; }
  }
  public virtual double Height
  {
    get { return height; }
    set { height = value; }
  }
}
```

```
public class Square: Rectangle {
  public override double Width
  {
    set {
      base.Width = value;
      base.Height = value;
    }
  }
  public override double Height
  {
    set {
      base.Height = value;
      base.Width = value;
    }
  }
}
```

Extensión y ocultación de métodos

- La diferencia entre `new` y `override` en un método en C# es que `new` oculta la implementación de la clase base y `override` la extiende.

Sin embargo, cuando la creación de una clase derivada provoca cambios en la clase base, es síntoma de un **mal diseño**.

El principio LSP pone en evidencia que la relación **es-un** tiene que ver con el comportamiento público extrínseco, del que los clientes dependen.

Ahora parece que funcionan `Square` y `Rectangle`, que matemáticamente quedan bien definidos.

Pero consideremos esto:

```
void g(Rectangle r)
{
  r.Width = 5;     // cree que es un Rectangle
  r.Height = 4;    // cree que es un Rectangle
  if(r.Area() != 20)
    throw new Exception("Bad area!");
}
```

¿Qué pasa si llamamos a `g(new Square(3))`?

El autor de `g` asumió que cambiar el ancho de un rectángulo deja intacto el alto. Si pasamos un cuadrado esto no es así.

Violación de LSP: Si pasamos una instancia de una clase derivada (`Square`), se altera el comportamiento definido por la clase base (`Rectangle`) de forma que `g` deja de funcionar.

¿Quién tiene la culpa?

¿Diseño o implementación?

Diseño vs implementación

¿Quién tiene la culpa?

- ¿El autor de `g` por asumir que "en un rectángulo su ancho y alto son independientes" (*invariante*)?
- ¿El autor de `Square` por violar el invariante?
- ¿De qué clase se ha violado el invariante? ¡De `Rectangle` y no de `Square`!

Para evaluar si un diseño es apropiado, no se debe tener en cuenta la solución por sí sola, sino en términos de los *supuestos razonables* que hagan los usuarios del diseño.