

Tema 3: Disseny

Anna Puig

Enginyeria Informàtica

Facultat de Matemàtiques i Informàtica,

Universitat de Barcelona

Curs 2021/2022



UNIVERSITAT DE
BARCELONA

Temari

1	Introducció al procés de desenvolupament del software	
2	Anàlisi de requisits i especificació	
3	Disseny	
4	Del disseny a la implementació	
5	Ús de frameworks de testing	
		3.1 Introducció
		3.2 Patrons arquitectònics
		3.3 Criteris de Disseny: G.R.A.S.P.
		3.4 Principis de Disseny: S.O.L.I.D.
		3.5 Patrons de disseny

[Articles de suport de cada principi](#)

3.4. Principis de Disseny: S.O.L.I.D.

Rígid

Immutable



Viscós

Fràgil

[Article resum de R.C. Martin \(fins la pàgina 18\)](#)

Agile Software Development, Principles, Patterns, and Practices

3.4. Principis de Disseny: S.O.L.I.D.

Principis de disseny [Robert C. Martin 98]:

- **S**: Single Responsibility Principle
- **O**: Open-Close Principle
- **L**: Liskov Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle



[Article resum de R.C. Martin \(fins la pàgina 18\)](#)

Agile Software Development, Principles, Patterns, and Practices

3.4. Principis de Disseny: S.O.L.I.D.

Single Responsibility Principle:

"One class should have one and only one responsibility"

De Marco 79 and Page-Jones 88



- Una classe ha de tenir **una i només una responsabilitat**
- Responsabilitat s'entén com a raó que fa canviar la classe
- Una classe hauria de tenir només una raó per a ser canviada
- Això permet tenir **ALTA** cohesió i evitar classes **fràgils**

3.4. Principis de Disseny: S.O.L.I.D.

Exemple de SRP: Modem

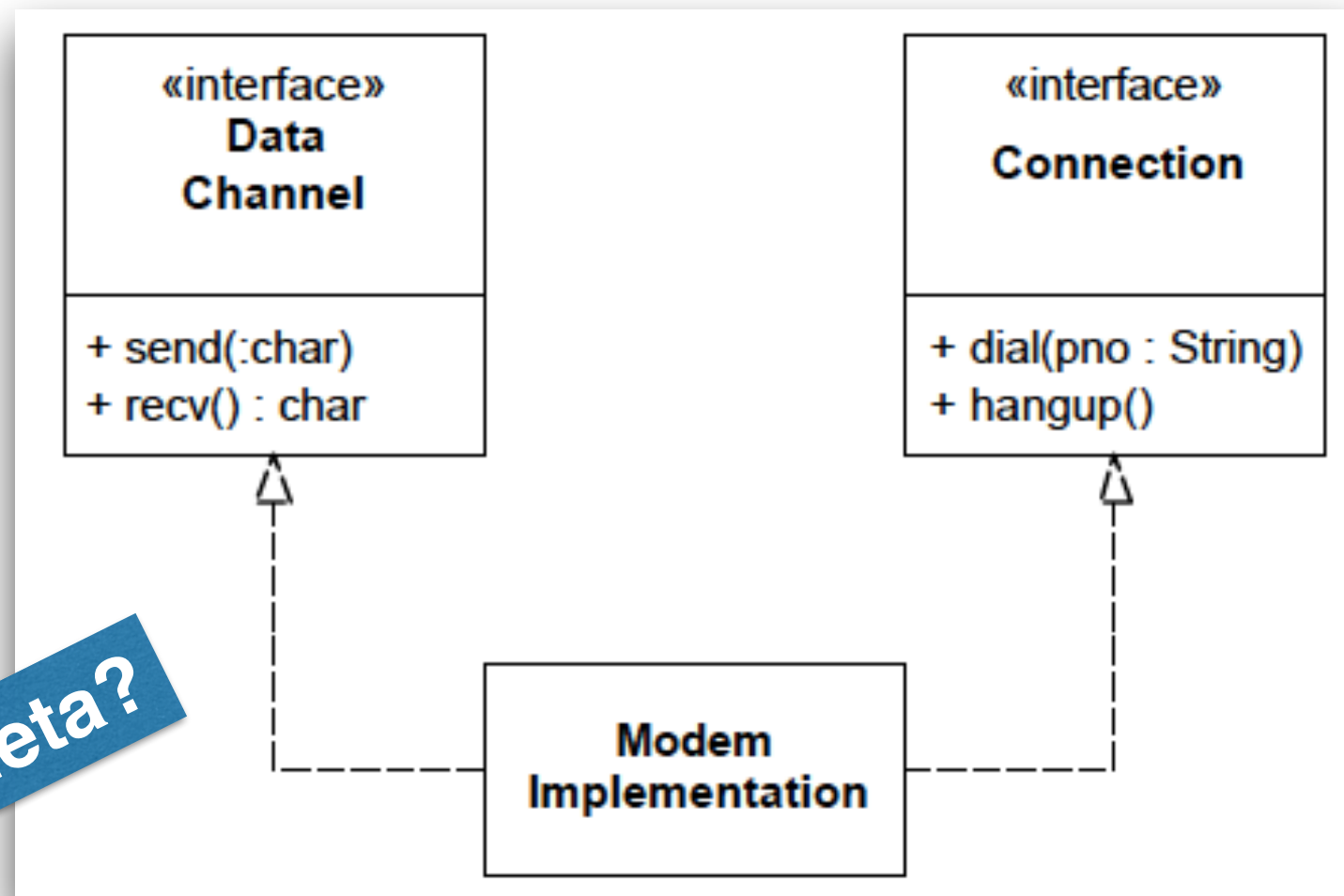
- Quants actors actuen?
- Quins d'ells tenen comportaments diferents?

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

3.4. Principis de Disseny: S.O.L.I.D.

- Cal identificar característiques que canvien per diferents raons i per a quines raons canvien
- Cal agrupar les característiques que canvien per les mateixes raons
- Per exemple,
 - usant diferents interfícies
 - usant diferents classes,...

Solució neta?

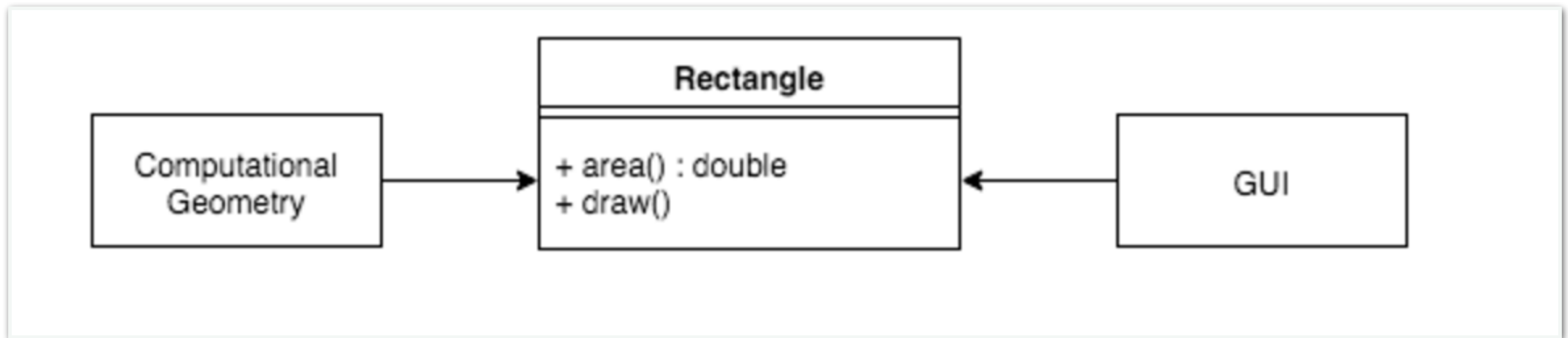


3.4. Principis de Disseny: S.O.L.I.D.

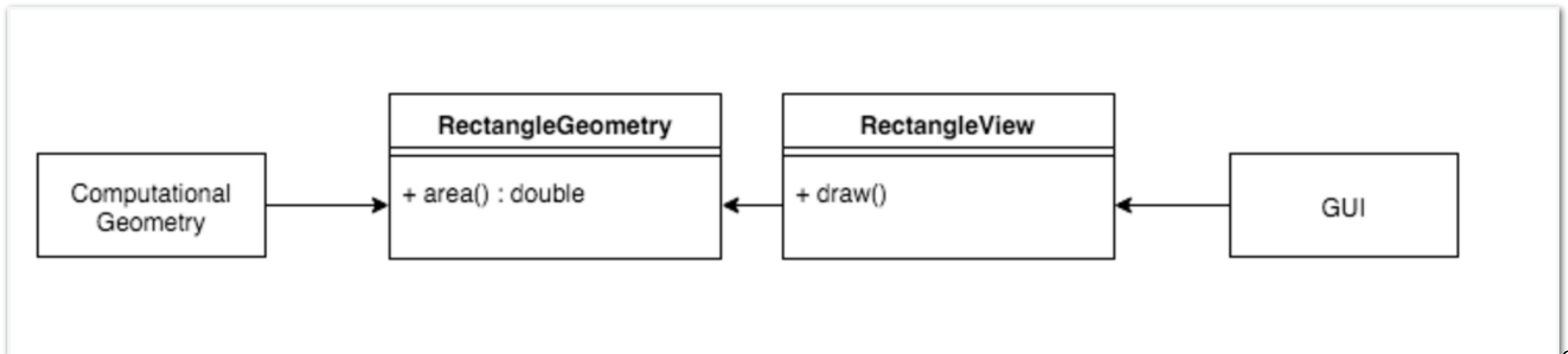
```
interface Employee
{
    public Pay calculate();
    public void report(Writer w);
    public void save();
    public void reload();
}
```

- Cal identificar característiques que canvien per diferents raons
- Cal agrupar les característiques que canvien per les mateixes raons

3.4. Principis de Disseny: S.O.L.I.D.



- Cal identificar característiques que canvien per diferents raons
- Cal agrupar les característiques que canvien per les mateixes raons



3.4. Principis de Disseny: S.O.L.I.D.

Open Closed Principle:

"Software components should be open for extension, but closed for modification"

Meyer, 88



- Els mòduls (classes, funcions, operacions, etc.) haurien de ser:
 - **Oberts** per extensió: per satisfer nous requisits
 - **Tancats** per modificació: L'extensió no implica canvis en el codi del mòdul. No s'ha de tocar la versió **executable** del mòdul.
- El comportament dels mòduls que satisfan aquest principi es canvia afegint nou codi, i no pas canviant codi existent.
- L'ús correcte del **polimorfisme** afavoreix aquest principi

3.4. Principis de Disseny: S.O.L.I.D.

- Segueix el principi obert-tancat?

```
void DrawAllShapes (ShapePointer list[], int n)
{
    for (int i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->type)
        {
            case square: DrawSquare ((struct Square*) s);
                          break;
            case circle: DrawCircle ((struct Circle*) s);
                          break;
        }
    }
}
```

3.4. Principis de Disseny: S.O.L.I.D.

- Segueix el principi obert-tancat?

```
public void draw(Shape[] shapes) {  
    for( Shape shape : shapes ) {  
        switch (shape.getType()) {  
            case Shape.SQUARE:  
                draw( (Square) shape) ;  
                break;  
            case Shape.CIRCLE:  
                draw( (Circle) shape) ;  
                break;  
        }  
    }  
}
```

3.4. Principis de Disseny: S.O.L.I.D.

- Ara si segueix el principi obert-tancat:

```
public void draw(Shape[] shapes) {  
    for( Shape shape : shapes ) {  
        shape.draw();  
    }  
}
```

Consell (R. Martin):

- fés que les coses que canvien sovint estiguin lluny de les que no canvien (estàtiques)
- si unes depenen de les altres, **les coses que canvien sovint** han de ser les que **depenen** de les que **no canvien**

3.4. Principis de Disseny: S.O.L.I.D.

- Què passa si es vol dibuixar primer tots els Cercles i després tots el Quadrats?

```
public void draw(List<Shape> shapes) {  
    for (Shape shape : shapes) {  
        if (shape.getClass() == Shape.Circle) {  
            shape.draw();  
        }  
    }  
    for (Shape shape : shapes) {  
        if (shape.getClass() == Shape.Square) {  
            shape.draw();  
        }  
    }  
}
```

3.4. Principis de Disseny: S.O.L.I.D.

- Què passa si es vol dibuixar primer tots els Cercles i després tots el Quadrats?

```
public void draw(List<Shape> shapes) {  
    shapes.sort(new ShapeComparator());  
    shapes.forEach(draw());  
}
```

```
public class ShapeComparator implements Comparator<Shape> {  
    @Override  
    public int compare(Shape p1, Shape p2) {  
        return (p1 instanceof Shape.Square);  
    }  
}
```

3.4. Principis de Disseny: S.O.L.I.D.

- Com comprometen el principi Obert-Tancat ...
 - Els atributs públics?

```
public class Device {  
    public boolean status;  
}
```

- Les variables globals?

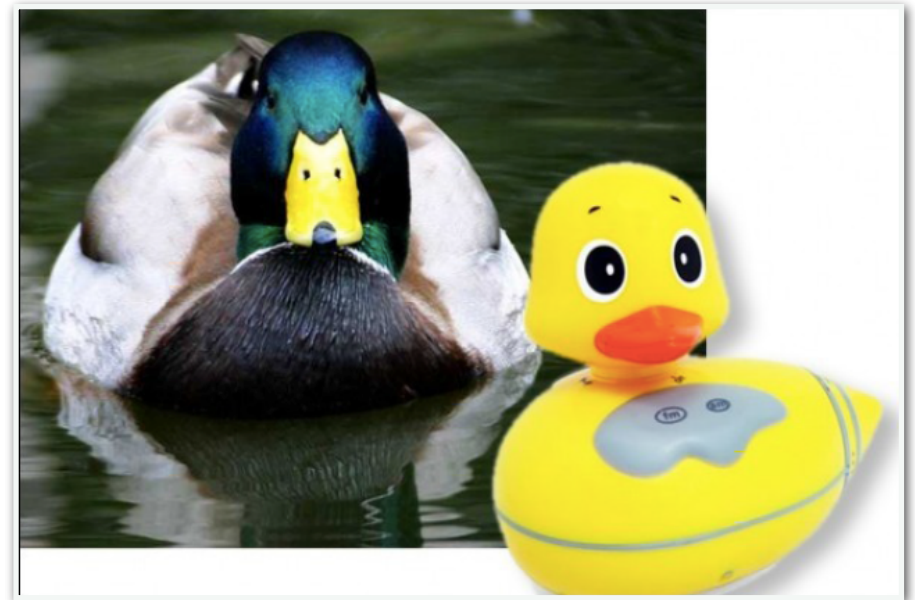
```
public class Time {  
    int hours; int minutes; int seconds;  
}
```


3.4. Principis de Disseny: S.O.L.I.D.

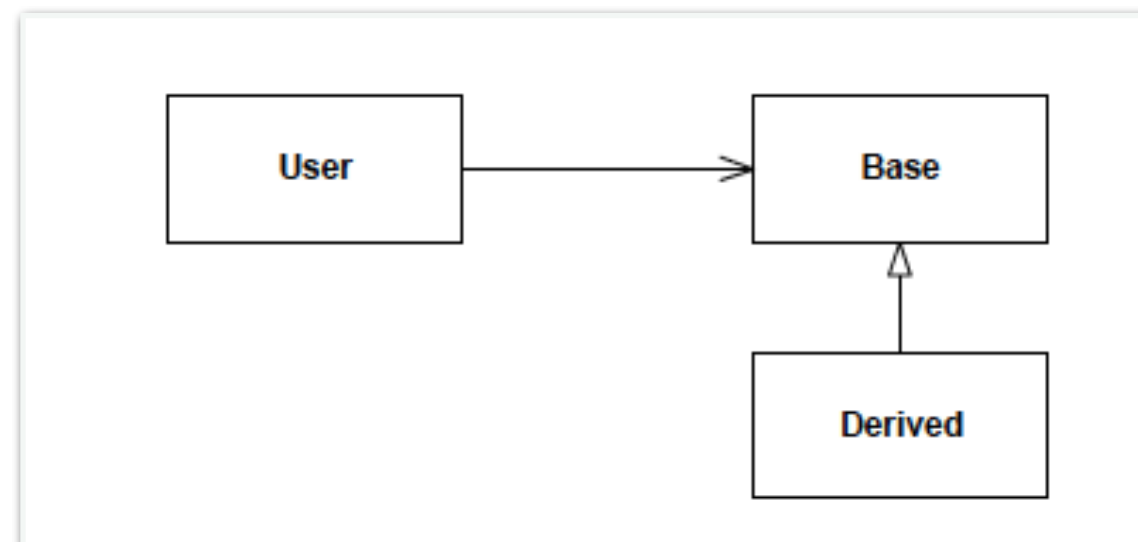
Liskov Substitution Principle:

"Derived types must be completely substitutable for their base types"

Barbra Liskov 1988

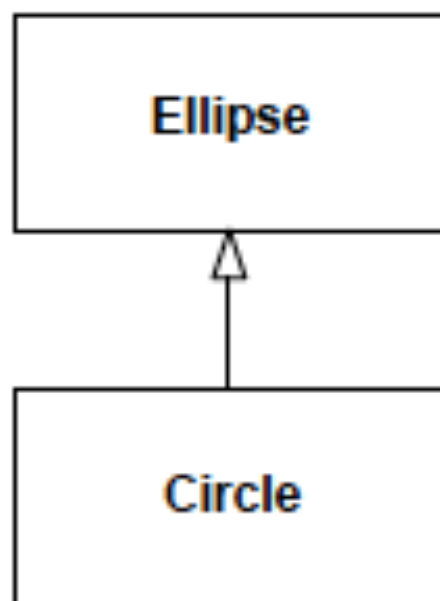


- Donada una entitat **Base** amb un cert mètode i altres subentitats **Derivades** que implementen el mètode original, el comportament d'un objecte **o1** que era de tipus Base no ha de variar si s'usa **o2** de tipus Derivat



3.4. Principis de Disseny: S.O.L.I.D.

Liskov Substitution Principle:



Ellipse
<ul style="list-style-type: none">- itsFocusA : Point- itsFocusB : Point- itsMajorAxis : double
<ul style="list-style-type: none">+ Circumference() : double+ Area() : double+ GetFocusA() : Point+ GetFocusB() : Point+ GetMajorAxis() : double+ GetMinorAxis() : double+ SetFoci(a:Point, b:Point)+ SetMajorAxis(double)

Què passa amb f quan e és de tipus Circle?

```
void f(Ellipse& e)
{
    Point a(-1,0);
    Point b(1,0);
    e.SetFoci(a,b);
    e.SetMajorAxis(3);
    assert(e.GetFocusA() == a);
    assert(e.GetFocusB() == b);
    assert(e.GetMajorAxis() == 3);
}
```

```
void Circle::SetFoci(const Point& a, const Point& b)
{
    itsFocusA = a;
    itsFocusB = a;
}
```

3.4. Principis de Disseny: S.O.L.I.D.

Liskov Substitution Principle:

```
void f(Ellipse& e)
{
    if (typeid(e) == typeid(Ellipse))
    {
        Point a(-1,0);
        Point b(1,0);
        e.SetFoci(a,b);
        e.SetMajorAxis(3);
        assert(e.GetFocusA() == a);
        assert(e.GetFocusB() == b);
        assert(e.GetMajorAxis() == 3);
    }
    else
        throw NotAnEllipse(e);
}
```

- Violacions en el LSP son violacions latents del Open-Closed Principle

3.4. Principis de Disseny: S.O.L.I.D.

Interface Segregation Principle:

"Clients should not be forced to implement unnecessary methods which they will not use"

Gamma et al., 1995



- No s'ha d'obligar a les classes a dependre de classes o mètodes que no han d'usar (deriva del SRP)
- Tot i que hi hagin classes molt grans que inclouen molts mètodes (interfícies no cohesionades), els clients (altres classes) només haurien de conèixer classes abstractes que tinguin interfícies cohesionades.
- Això permet tenir **ALTA** cohesió i evitar classes **fràgils**

3.4. Principis de Disseny: S.O.L.I.D.

Interface Segregation Principle:

```
interface IWorker {
    public void work();
    public void eat();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
    public void eat() {
        // ..... eating in launch
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }

    public void eat() {
        //.... eating in launch break
    }
}
```

```
class Manager {
    IWorker worker;

    public void setWorker(IWorker w) {
        worker=w;
    }

    public void manage() {
        worker.work();
    }
}
```

I si afegim un Robot que no menja?

3.4. Principis de Disseny: S.O.L.I.D.

Interface Segregation Principle:

```
interface IWorker extends IFeedable, IWorkable {  
}  
  
interface IWorkable {  
    public void work();  
}  
  
interface IFeedable {  
    public void eat();  
}  
  
class Worker implements IWorkable, IFeedable {  
    public void work() {  
        // ....working  
    }  
  
    public void eat() {  
        //.... eating in launch break  
    }  
}  
  
class Robot implements IWorkable {  
    public void work() {  
        // ....working  
    }  
}
```

3.4. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

"Depend on abstractions, not on concretions"

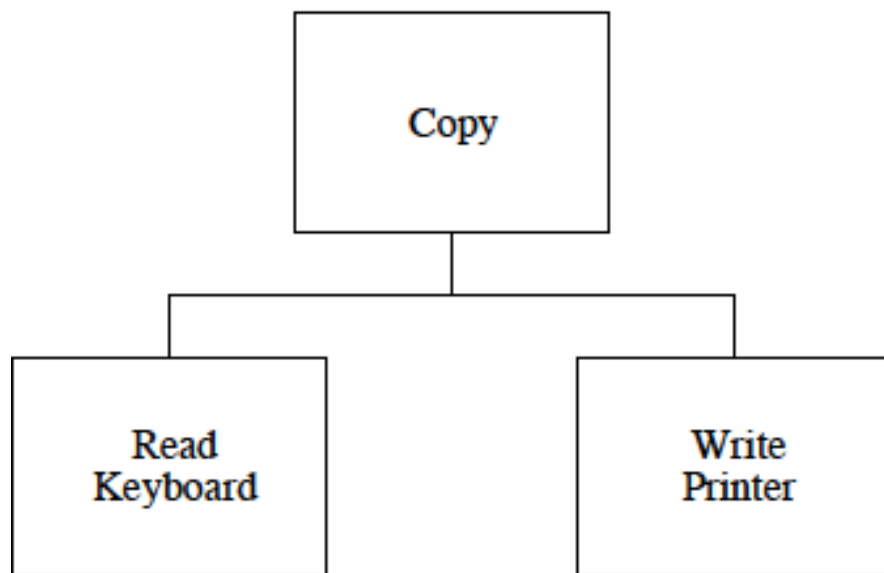
Martin, 1997



- Les classes d'alt nivell no han de canviar per que canvien les classes més senzilles o de baix nivell. Les dues haurien de dependre d'abstraccions.
- Les abstraccions no han de dependre de detalls tecnològics, són els detalls que haurien de dependre de les abstraccions.
- Aplicar aquest principi dóna **BAIX** acoblament

3.4. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle?



```
void Copy()  
{  
    int c;  
    while ((c = ReadKeyboard()) != EOF)  
        WritePrinter(c);  
}
```

I si el volem estendre per a guardar a disc?

3.4. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle?

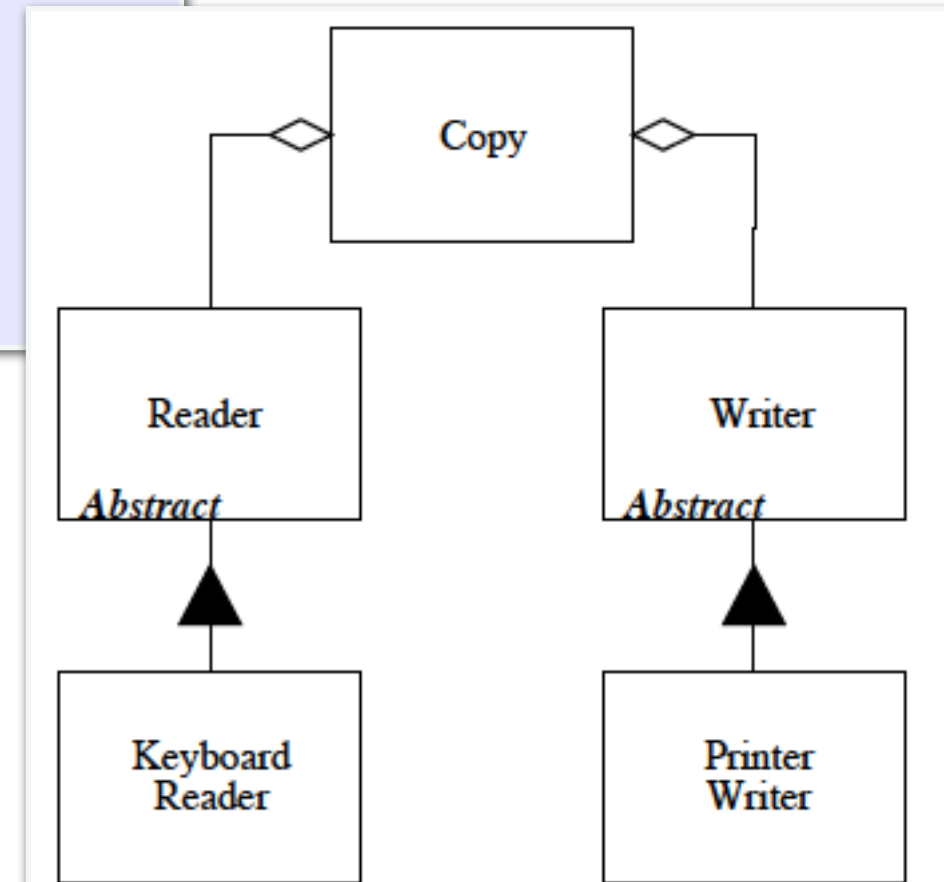
```
enum OutputDevice {printer, disk};

void copy(OutputDevice dev)
{
    int c;
    while((c=readKeyboard()) != EOF)
    {
        if (dev == printer)
            writePrinter(c);
        else
            writeDisk(c);
    }
}
```

3.4. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

```
void copy(Reader input, Writer output)
{
    int c;
    while( (c=input.read()) != EOF) {
        output.write(c);
    }
}
```



3.4. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

- Com es creen les classes dels diferents tipus concret?
- Mitjançant el patró d'**Injecció de Dependències**, es subministren objectes a la classe, en lloc de crear-los dins de la mateixa classe.

```
public class Vehiculo {  
  
    private Motor motor = new Motor16Valvules();  
  
    /** @retorna la velocidad del vehículo*/  
    public Double enAceleracionDePedal(int presionDePedal) {  
        motor.setPresionDePedal(presionDePedal);  
        int torque = motor.getTorque();  
        Double velocidad = ... //realiza el cálculo  
        return velocidad;  
    }  
  
}
```

3.4. Principis de Disseny: S.O.L.I.D.

Dependency Inversion Principle:

```
public class Vehiculo {  
  
    private Motor motor = null;  
  
    public void setMotor(Motor motor){  
        this.motor = motor;  
    }  
  
    /** @retorna la velocidad del vehículo*/  
    public Double enAceleracionDePedal(int presionDePedal) {  
        Double velocidad = null;  
        if (null != motor){  
            motor.setPresionDePedal(presionDePedal);  
            int torque = motor.getTorque();  
            velocidad = ... //realiza el cálculo  
        }  
        return velocidad;  
    }  
}
```

```
public class VehiculoFactory {  
  
    public Vehiculo construyeVehiculo() {  
        Vehiculo vehiculo = new Vehiculo();  
        Motor motor = new Motor16Valvules();  
        vehiculo.setMotor(motor);  
        return vehiculo;  
    }  
}
```