Design principles

What makes great software ?

What makes great software ?

What makes great software ?

Works

No bugs

Easy to maintain

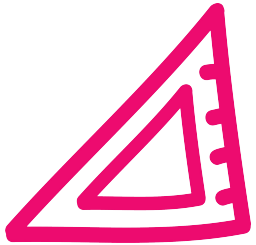
Easy to add new features

Good performances

What makes great software ?

Make it work

1



Make it clean

2

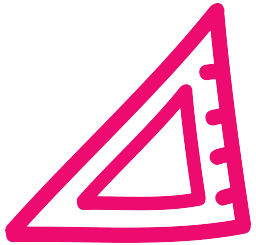
Make it fast

3

What makes great software ?

Make it work

1



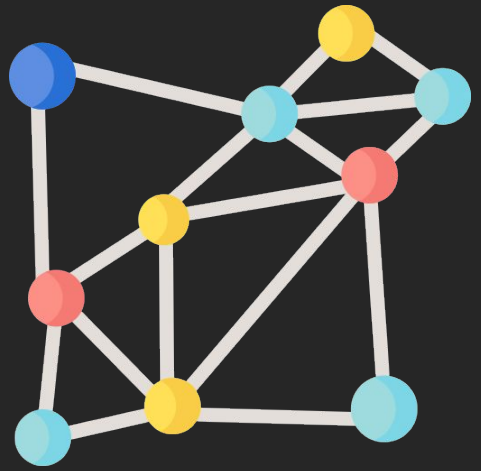
Make it clean

2

Make it fast

3

Cohesion / Coupling



Definition

Cohesion & **coupling** are abstract, complementary **measures**

Definition

Cohesion & **coupling** are abstract, complementary **measures**

- Cohesion
  - How much a piece of code is tailored for one **coherent** thing.
  - How much one class (or method) has a **Single responsibility**
    - Less is more

Definition

Cohesion & **coupling** are abstract, complementary **measures**

- Cohesion
  - How much a piece of code is tailored for one **coherent** thing.
  - How much one class (or method) has a **Single responsibility**
    - Less is more
- Coupling
  - How much a piece of code depends on the surrounding code.
  - How much **“changing a class implies changing others”**.

Definition

Cohesion & **coupling** are abstract, complementary **measures**

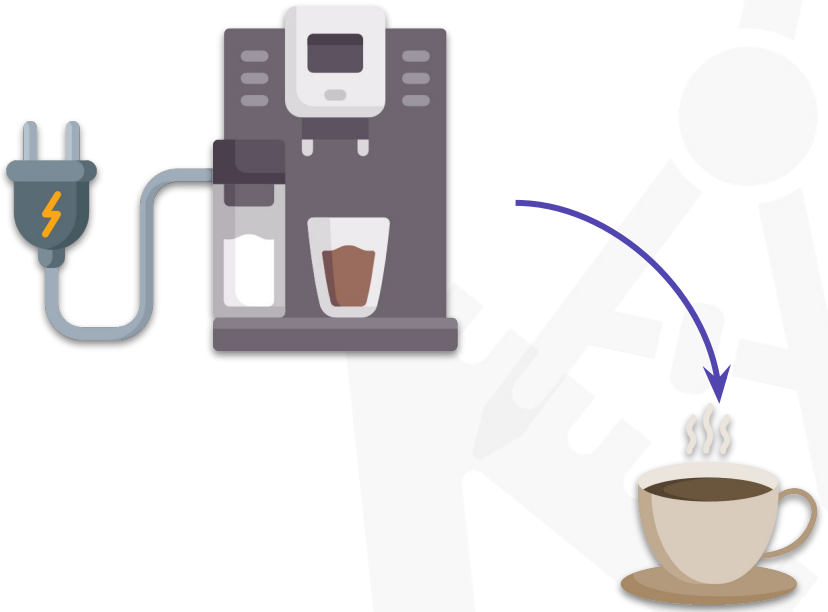
- Cohesion
  - How much a piece of code is tailored for one **coherent** thing.
  - How much one class (or method) has a **Single responsibility**
    - Less is more
- Coupling
  - How much a piece of code depends on the surrounding code.
  - How much **“changing a class implies changing others”**.

You should seek **high cohesion** & **loose coupling** for good software **maintainability**

Cohesion

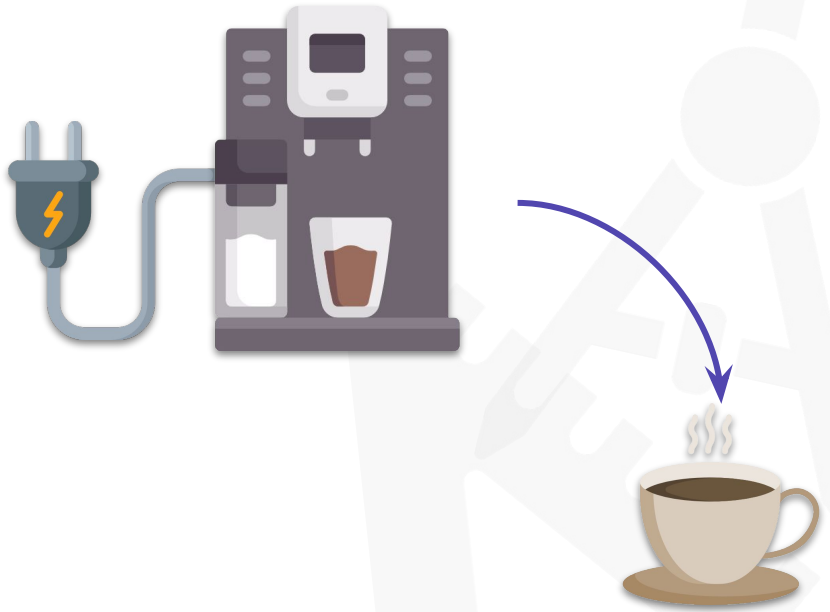
Cohesion

Example: brew coffee with an automatic coffee machine



Cohesion

Example: brew coffee with an automatic coffee machine



- ⊕ – Easy to use
  - press a single button
- – Standalone
  - Does not depend on other devices
- ⊖ – Big machine
  - expensive to buy
- – Hard to repair
  - if some part fail, may need to buy a new one.

Cohesion

Example: brew coffee with an automatic coffee machine

- Easy to use
  - call a single method
- Standalone
  - does not depend on other classes

- Big code
  - expensive to write
- Hard to refactor
  - if some part fail, may need to re-write everything

```
public class AutomaticCoffeeMachine {
    public Coffee brew() {
        var water = this.dispenseWater();
        var hotWater = this.preHeat(water);
        var beans = this.getCoffeeBean();
        var ground = this.grindCoffee(beans);
        var puck = this.tamp(ground);
        hotWater = this.boil(hotWater);
        return this.infuse(hotWater, puck);
    }

    private Water dispenseWater() { /* ... */ }
    private Water preHeat() { /* ... */ }
    private Water boil() { /* ... */ }
    private Ground grindCoffee() { /* ... */ }
    private Puck tamp() { /* ... */ }
    private Coffee infuse() { /* ... */ }
}
```

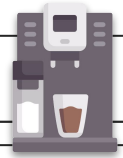
Cohesion

Example: brew coffee with an automatic coffee machine

– Loose coupling
- no dependencies

– Low cohesion
- do too many things
- device too complicated

```
AutomaticCoffeeMachine
```



```
+ brew(): Coffee
- dispenseWater(): Water
- preHeat(water: Water): Water
- getCoffeeBean(): Beans
- grindCoffee(beans: Beans): Ground
- tampCoffee(ground: Ground): Puck
- boil(water: Water): Water
- infuse(w: Water, p: Puck): Coffee
```

Cohesion: how ?

Increase cohesion ?

```
public class AutomaticCoffeeMachine {
    public Coffee brew() {
        var water = this.dispenseWater();
        var hotWater = this.preHeat(water);
        var beans = this.getCoffeeBean();
        var ground = this.grindCoffee(beans);
        var puck = this.tamp(ground);
        hotWater = this.boil(hotWater);
        return this.infuse(hotWater, puck);
    }

    private Water dispenseWater() { /* ... */ }
    private Water preHeat() { /* ... */ }
    private Water boil() { /* ... */ }
    private Ground grindCoffee() { /* ... */ }
    private Puck tamp() { /* ... */ }
    private Coffee infuse() { /* ... */ }
}
```

Cohesion: how ?

Increase cohesion ?

- split classes into smaller classes
  - use **composition & aggregation**

```
public record AutomaticCoffeeMachine (
        WaterSource waterSource,
        Heater heater,
        BeanSource beanSource,
        Grinder grinder,
        Tamper tamper
    ) {

    public Coffee brew() {
        var water = waterSource.getWater();
        var hotWater = heater.preHeat(water);
        var beans = beanSource.getCoffeeBean();
        var ground = grinder.grind(beans);
        var puck = tamper.tamp(ground);
        hotWater = heater.boil(hotWater);
        return this.infuse(puck, hotWater);
    }
   private Coffee infuse() { /* ... */ }
}
```

Cohesion: how ?

Increase cohesion ?

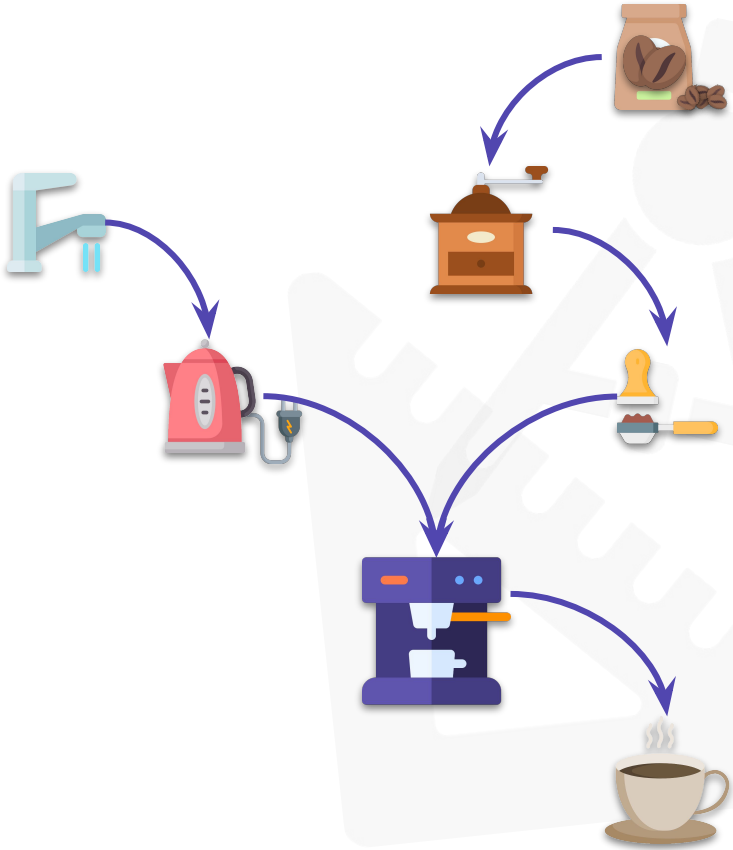
- split classes into smaller classes
  - use **composition & aggregation**
- Move boilerplate code
  - or remove it with **Aspect-Oriented Programming**

```
public record AutomaticCoffeeMachine (
        WaterSource waterSource,
        // Heater heater,
        BeanSource beanSource,
        Grinder grinder,
        Tamper tamper
    ) {
    @PreHeat()
    public Coffee brew() {
        var hotWater = waterSource.getWater();
        // var hotWater = heater.preHeat(water)
        var beans = beanSource.getCoffeeBean();
        var ground = grinder.grind(beans);
        var puck = tamper.tamp(ground);
        // hotWater = heater.boil(hotWater);
        return this.infuse(puck, hotWater);
    }
  private Coffee infuse() { /* ... */ }
}
```

Coupling

Coupling

Example: brew coffee with a manual coffee machine



Coupling

Example: brew coffee with a manual coffee machine

⊕
- Specialized devices
  - Can be reused
  - Easy to fine-tune
- Parts can be replaced individually
  - Easier to evolve

⊖
- Hard to use
  - executes in a precise sequence
- Higher risk of failure
  - if one device fails, no coffee

Coupling

Example: brew coffee with a manual coffee machine

- Specialized classes
  - Can be reused
  - Easy to customize
- Can be re-implemented partially
  - Easier to evolve

- Hard to use
  - configure everything in correct order
- Higher risk of failure
  - if one class fails, no coffee

```
// init components
var ws = new WaterSource();
var cbs = new CoffeeBeanSource();
var h = new Heater(95);
var g = new Grinder(0.35);
var t = new Tamper(400);
var cp = new CoffeePress(9.5, 60);

// prepare
var water = ws.getWater("arabica");
var beans = cbs.getCoffeBeans();
var ground = g.grind(beans);
var puck = t.tamp(ground);
water = h.preHeat(water);
var hotWater = h.boil(water);

// brew coffee
coffee = new cp.brew(hotWater, puck);
```

Coupling

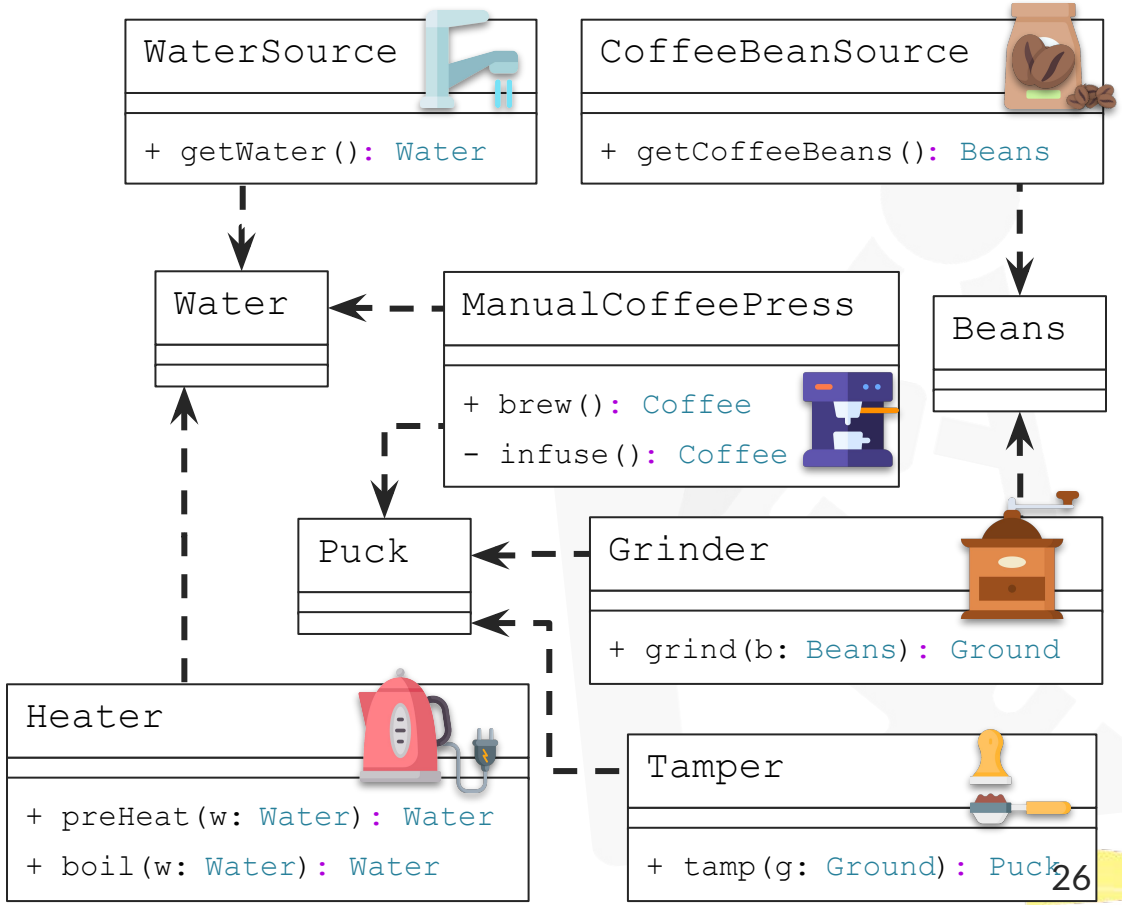
Example: brew coffee with a manual coffee machine

⊕ – High cohesion

- Small classes
- Easy to implement
- **Single responsibility**

⊖ – Tight coupling

- High dependency on each other
- High risks a change induces some changes to other classes



Coupling: How ?

Loosen coupling ?

```
var machine = new CoffeeMachine();

// BAD ===============
// brew() is tightly coupled with specific
ingredients.
coffee = machine.brew(
   hotWater, ground, milk,
   sugar, caramel, cacao
);
```

Coupling: How ?

– reduce the number of parameters

```
var machine = new CoffeeMachine();

// GOOD: Less parameters ============
// brew() is coupled with an abstract recipe.
var recipe = new Recipe(
   hotWater, ground, milk,
   sugar, caramel, cacao
);
coffee = machine.brew(recipe);
```

Coupling: How ?

– reduce the number of parameters

```
// BAD ============
// Should not create new ingredients
// for each coffee we make
var ws = new WaterSource();
var bs = new BeanSource();
var ms = new MilkSource();

// Recipe is tightly coupled with ingredients
var latteReceipe = new Receipe(
   ws.getWayer, bs.getBeans(), ms.getMilk(),
   null, // no sugar
   null, // no caramel
   null, // no cacao
);

// not easy to brew a latte
var latteMacciato =
     machine.brew(latteReceipe);
```

Coupling: How ?

- reduce the number of parameters
- Use design patterns
  - facade, singleton, builder, ...

```
// GOOD: Use design patterns ============
// Facade
class LatteMacciatoFacade {
 Coffee brew() {
   // Singletons
   var w = WaterSource.INSTANCE.getWater()
   var b = CoffeeSource.INSTANCE.getBeans();
   var m = MilkSource.INSTANCE.getMilk();

     // builder
     var rb = new ReceipeBuilder()
          .water(w)
          .beans(b)
          .milk(m);

     return this.machine.brew(rb.build())
   }
}
```

Coupling: How ?

- reduce the number of parameters
- Use design patterns
  - facade, singleton, builder, …

```
// BAD =======
// Depend on many method calls
grinder.setLevel(Level.FINE);
grinder.setBeans(beans);
grinder.start();
var weight = 0;
while (weight < CoffeeWeight.ESPRESSO) {
   weight += grinder.continue();
}
grinder.stop();

var ground = grinder.getCoffeeGround();
```

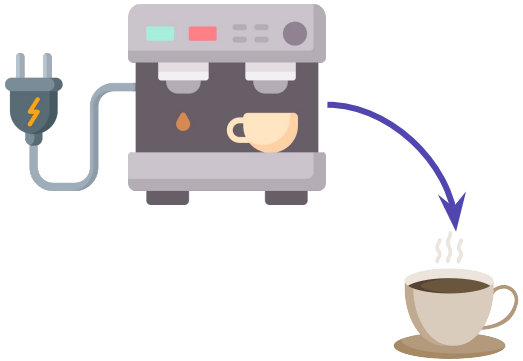
Coupling: How ?

- reduce the number of parameters
- Use design patterns
  - facade, singleton, builder, …
- reduce the number of method calls

```
// GOOD ======
// Wrap dependents instructions
// into a single method
Ground grind(CoffeeBeans beans) {
   grinder.setLevel(Level.FINE);
   grinder.setBeans(beans);
   grinder.start();
   var weight = 0;
   while (weight < CoffeeWeight.ESPRESSO) {
       weight += grinder.continue();
   }
   grinder.stop();

   var ground = grinder.getCoffeeGround();
   return ground;
}
```

A matter of trade of...

- Loose coupling ⊕
- Poor cohesion ⊖



- Tight coupling ⊖
- Good cohesion ⊕

Seek high **cohesion** and **loose coupling**

Adjusting between cohesion and coupling is often a **tradeoff**:

- less classes:
  - bigger classes ⊖
  - less cohesion ⊖
  - loose coupling ⊕
- more classes
  - smaller classes ⊕
  - more cohesion ⊕
  - tight coupling ⊖

- Design patterns for less coupling
  - Builder, Singleton, Facade, Strategy, …
- Design patterns for more cohesion
  - Proxy, …

SOLID

A rule of thumbs

S.O.L.I.D

Single Responsibility

O

L

I

D

Single Responsibility

Open / Close principle

L

I

D

Single Responsibility

Open / Close principle

Liskov Substitution

I

D

Single Responsibility

Open / Close principle

Liskov Substitution

Interface Segregation

D

Single Responsibility

Open / Close principle

Liskov Substitution

Interface Segregation

Dependency Inversion

Single Responsibility

- One responsibility per class / method
  - Better cohesion
  - Smaller class, easier to implement

Single Responsibility

DON'T

```
class AutomaticCoffeeMachine {
   private Beans getBeans() { }
   private Ground grind(Beans beans) { }
   private Water getWater() { }
   private Water boil(Water water) { }
   private Puck tamp(Ground ground) { }
   public Coffee brew(Water w, Puck p) { }
}
```

DO

SOLID

Single Responsibility

DON'T

```
class AutomaticCoffeeMachine {
   private Beans getBeans() { }
   private Ground grind(Beans beans) { }
   private Water getWater() { }
   private Water boil(Water water) { }
   private Puck tamp(Ground ground) { }
   public Coffee brew(Water w, Puck p) { }
```

```
}
```

DO (Cohesion)

```
class Grinder {
   private Ground grind(Beans beans) { }
}
```

```
class Heater {
   private Water boil(Water water) { }
}
```

```
class Tamper {
   private Puck tamp(Ground ground) { }
}
```

```
class CoffeePress {
   public Coffee brew(Water w, Puck puck) { }
}
```

Open / Close Principle

- Open for extensions
  - Should be easy to add new features
- Closed for modifications
  - Adding new features should not change existing code.

Open / Close Principle

DON'T

```
enum CoffeeType {
  RISTRETTO, ESPRESSO, AMERICANO
}
class CoffeePress {
   public Coffee brew(CoffeeType type) {
       float duration;
       if (type == CoffeeType.RISTRETTO)
           duration = 20;
       if (type == CoffeeType.ESPRESSO)
           duration = 30;
       if (type == CoffeeType.AMERICANO)
           duration = 40;
   }
}
```

DO

Open / Close Principle

DON'T (Coupling)

```
enum CoffeeType {
  RISTRETTO, ESPRESSO, AMERICANO
}
class CoffeePress {
   public Coffee brew(CoffeeType type) {
       float duration;
       if (type == CoffeeType.RISTRETTO)
           duration = 20;
       if (type == CoffeeType.ESPRESSO)
           duration = 30;
       if (type == CoffeeType.AMERICANO)
           duration = 40;
   }
}
```

DO

```
enum CoffeeType {
   RISTRETTO (20),
   ESPRESSO (30),
   AMERICANO (40);

   private float d;
   CoffeeType(float d) { this.d = d; }
   float getDuration() { return this.d; }
}

class CoffeePress {
   public Coffee brew(CoffeeType type) {
       float duration = type.getDuration();
   }
}
```

Liskov Substitution

- Let *T* a type, and *S* a subtype of *T*
- If *q(T)* is a property of *T*, this property has to be true for *S* as well
  - q(T) => q(S)
- At any time, a type *T* can be replaced with a subtype *S* without breaking the code.

SOLID

Liskov Substitution

```
interface Beans {}

class ArabicaBeans implements Beans {
}
class RobustaBeans implements Beans {
}


class Grinder {
   Ground grind(Beans b     ) {
       /* ... */  }
}

new Grinder().grind(new ArabicaBeans());
```

Liskov Substitution

DON'T

```
interface Beans {}

class ArabicaBeans implements Beans {
}
class RobustaBeans implements Beans {
}
class FrenchBeans implements Beans {
}
class Grinder {
   Ground grind(Beans b) {
       /* ... */  }
}

new Grinder().grind(new FrenchBeans());
```

DO

SOLID

Liskov Substitution

DON'T (Bad substitution)

```
interface Beans {}

class ArabicaBeans implements Beans {
}
class RobustaBeans implements Beans {
}
class FrenchBeans implements Beans {
}
class Grinder {
   Ground grind(Beans b) {
      /* ... */  }
}

new Grinder().grind(new FrenchBeans());
```

DO (Good abstraction)

```
interface Beans {}
interface CoffeeBeans extends Beans {}
class ArabicaBeans implements CoffeeBeans {
}
class RobustaBeans implements CoffeeBeans {
}
class FrenchBeans implements Beans {
}
class Grinder {
   Ground grind(CoffeeBeans b) {
   /* ... */ }
}

new Grinder().grind(new ArabicaBeans());
```

Interface Segregation

- Large interfaces offer a lot of services
  - Hard to repurpose
- Turn large interfaces into a composition of smaller ones

Interface Segregation

- Large interfaces offer a lot of services
  - Hard to repurpose
- Turn large interfaces into a composition of smaller ones

```
class Hammer implements Crowbar, Mallet { }
```

- maintainability
- testability
- reusability

Interface Segregation

DON'T

```
interface CoffeeGround {
   void tamp();
   void infuse();
   void dose();
}

interface TeaBag {
   void infuse();
   void dose();
}
```

DO

SOLID

Interface Segregation

DON'T (large interfaces)

```
interface CoffeeGround {
   void tamp();
   void infuse();
   void dose();
}

interface TeaBag {
   void infuse();
   void dose();
}
```

DO (reusable interfaces)

```
interface Tampable {
   void tamp();
}
interface Infusable {
   void infuse();
}
interface Dosable {
   void dose();
}

interface CoffeeGround
      extends Tampable, Infusable, Dosable {    }

interface TeaBag
      extends Infusable, Dosable {    }
```

Dependency inversion

Depend on abstraction, not implementations

Eg:

- “I need a screwdriver”

Not

- “I need the “blue, stainless steel, phillips screwdriver”

- Loosely coupled on implementation details
- Change with other implementations

SOLID

Dependency inversion

DON'T

```
interface CoffeeBeans {}

class ArabicaBeans implements Beans {  }
class RobustaBeans implements Beans {  }

ArabicaBeans beans = new ArabicaBeans();
new Grinder().grind(beans);

class Grinder {
  Ground grind(ArabicaBeans beans ) {
  /* ... */ }
}
```

DO

SOLID

Dependency inversion

DON'T (depend on concrete types)

```
interface CoffeeBeans {}

class ArabicaBeans implements CoffeeBeans {
}
class RobustaBeans implements CoffeeBeans {
}

ArabicaBeans beans = new ArabicaBeans();
new Grinder().grind(beans);

class Grinder {
  Ground grind(ArabicaBeans beans     ) {
  /* ... */ }
}
```

DO

```
interface CoffeeBeans {}

class ArabicaBeans implements CoffeeBeans {
}
class RobustaBeans implements CoffeeBeans {
}

Beans beans = new ArabicaBeans();
new Grinder().grind(beans);

class Grinder {
  Ground grind(CoffeeBeans beans        ) {
  /* ... */ }
}
```

- **S**ingle Responsibility
- **O**pen / Close principle
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

Other best practises

DRY, KISS, YAGNI

DRY

Don’t **R**epeat **Y**ourself

- Do not copy-paste (large) pieces of codes
- Factorize code
  - With composition
  - With inheritance

Best practises

DRY

DON'T

```
class Kettle {
    // ...
    Water boil(Water water) {
        while(water.temperature < 100)
            water.temperature++;
    }
}
class CoffeeMachine {
    // ...
    Water boil(Water water) {
        while(water.temperature < 100)
            water.temperature++;
    }
}
```

DO

Best practises

DRY

DON'T

```
class Kettle {
    // ...
    Water boil(Water water) {
        while(water.temperature < 100)
            water.temperature++;
    }
}
class CoffeeMachine {
    // ...
    Water boil(Water water) {
        while(water.temperature < 100)
            water.temperature++;
    }
}
```

ALSO DON'T

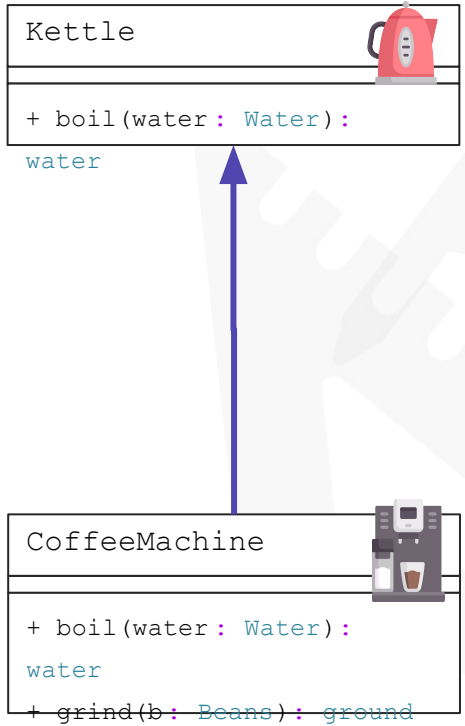
```
class Kettle {
  // ...
  Water boil(Water water) {
      while(water.temperature < 100)
          water.temperature++;
  }
}
class CoffeeMachine extends Kettle {

}
```

DRY

DON'T

```
class Kettle {
    // ...
    Water boil(Water water) {
        while(water.temperature < 100)
            water.temperature++;
    }
}
class CoffeeMachine {
    // ...
    Water boil(Water water) {
        while(water.temperature < 100)
            water.temperature++;
    }
}
```

ALSO DON'T

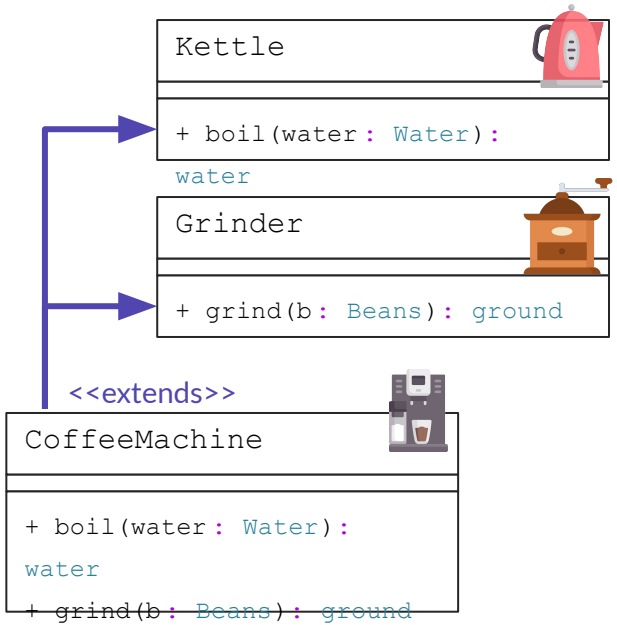


Prefer composition over inheritance

- Inheritance:
  - Child class is tightly coupled to its parent
  - Inherit from multiple classes causes problems

Prefer composition over inheritance

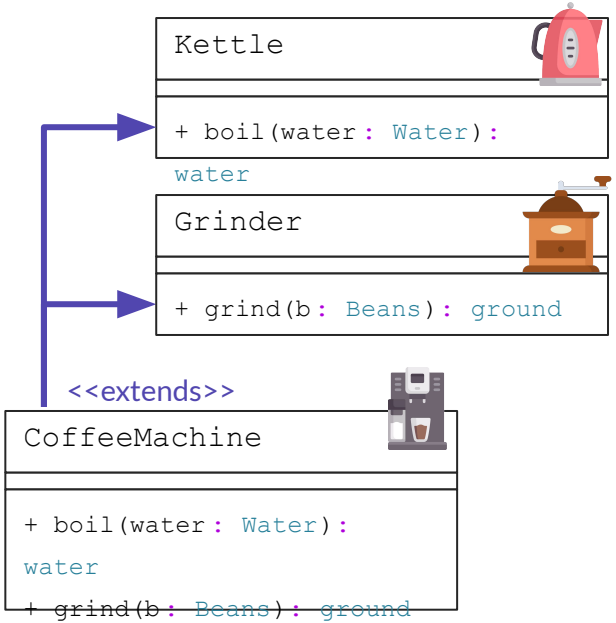
DON'T



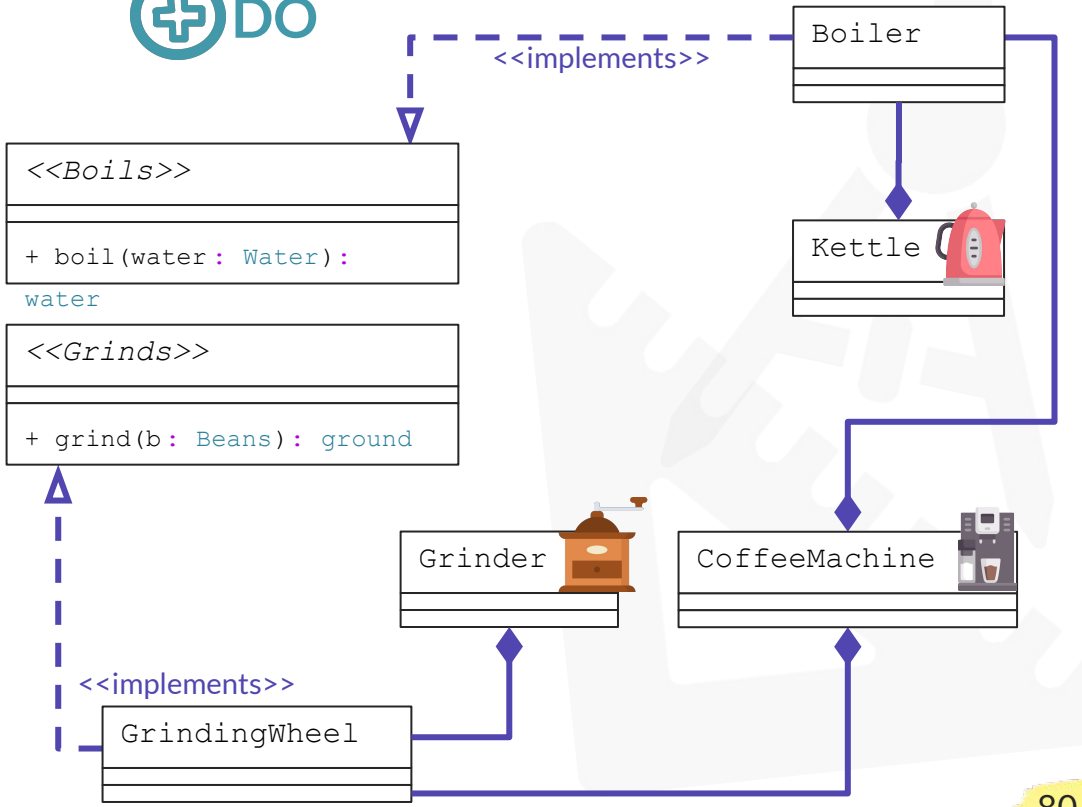
DO

Prefer composition over inheritance

DON'T



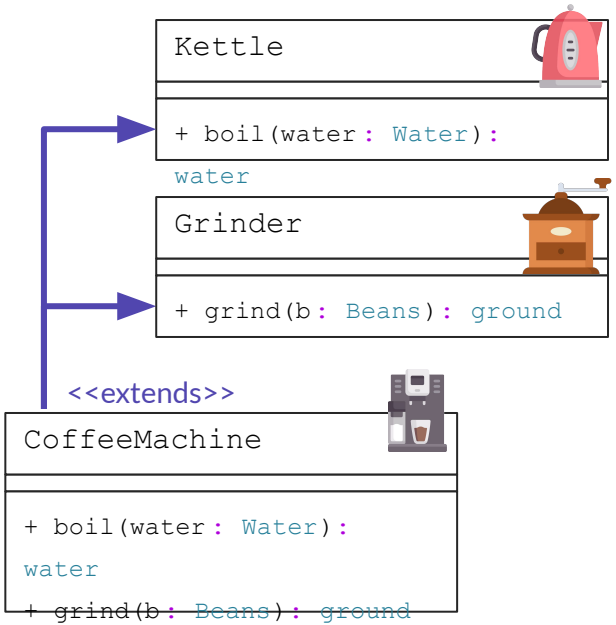
DO



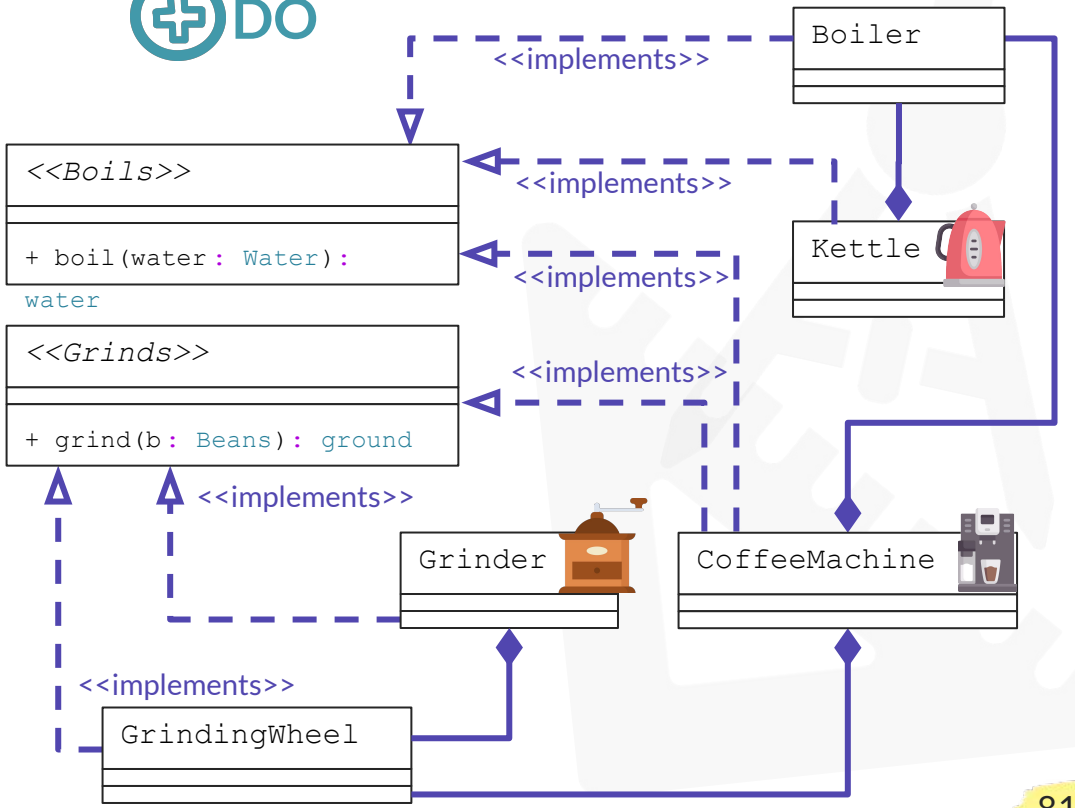
Best practises

Prefer composition over inheritance

DON'T



DO



- Seek high cohesion

- The cohesion of a class / method usually decreases when its size increase.

Thank you.

Contributors

- Nicolas THIERION <nthierion@takima.f>
- Maxime BIBOS <mbibos@takima.fr>

Leave feedback:
https://forms.gle/J8jAX4HJdEfaDgydA

See also

- P10 - Computers foundations
- J10 - Introduction to java
- P20 - OOP basics
- D05 - Modelization with UML
- J20 - Java basics
- J30 - Java collections
- J40 - Concurrency
- P30 - Functional Programming
- D20 - Design patterns

Contact formation@takima.fr

TODO

YAGNI

© Crédits

Icons found on
https://www.flaticon.com

- Freepik :
  - coffee machine, power plug, coffee press, kettle, coffee grinder, compass, ruler, coffee beans, french beans, crowbar, mallet