SOLID

Programming 2.2

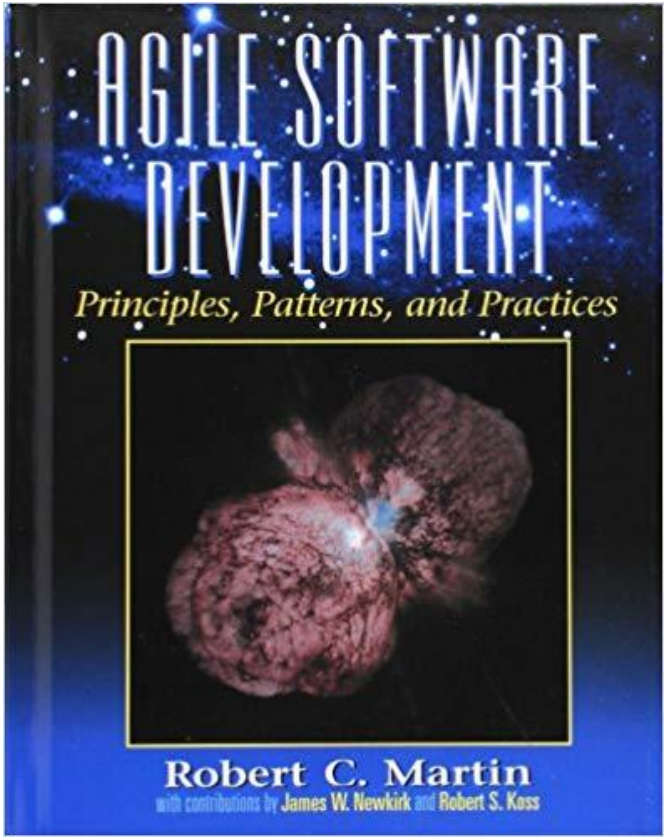
KdG Karel de Grote Hogeschool

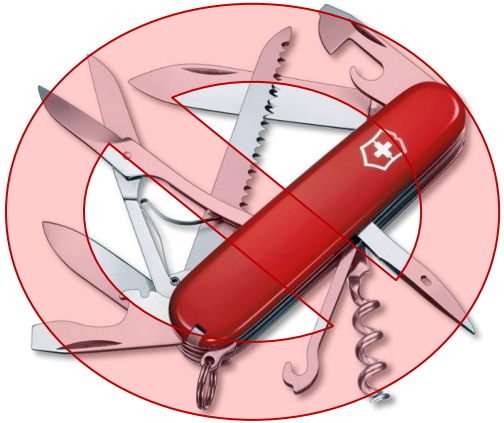
Agenda

S	ingle Responsibility
O	pen / Closed
L	iskov Substitution
I	nterface Segregation
D	ependency Inversion

SOLID

- OO design patterns/principles with a focus on interfaces
- Patterns/principles are guidelines, not universal laws: evaluate the context to decide their applicability
- Do not over-design: you can always refactor when needed





Single Responsibility Principle

Single Responsibility Principle (SRP)

There should only be one reason to change a class

- Related principles
  - High cohesion
  - Separation of Concerns
- If a class has more responsibilities
  - It is more complex and harder to maintain
  - A change for responsibility A can have an adverse impact on responsibility B
  - If a class changes for responibility A, those who use it for responsibility B will have to adapt to the new class, without getting any benefit.

Single Responsibility Principle

```
public class UserAccount {

  public void changePhone(String phone) {
    if (checkAccess()) {
      this.phone=phone;
    }
  }

  public boolean checkAccess() {
    return getEmail().endsWith("kdg.be");
  }

}
```

User Profile

Security

Single Responsibility Principle

```
public class UserAccount {
  public void changePhone(String phone){
    if (securitySvc.checkAccess(this)){
      this.phone=phone;
    }
  }
```

User Profile

```
public class SecurityService {
  public boolean checkAccess(UserAccount user) {
      return user.getEmail().endsWith("kdg.be");
    }
}
```

Security

KdG
Karel de Grote Hogeschool

Single Responsibility Principle / CQS

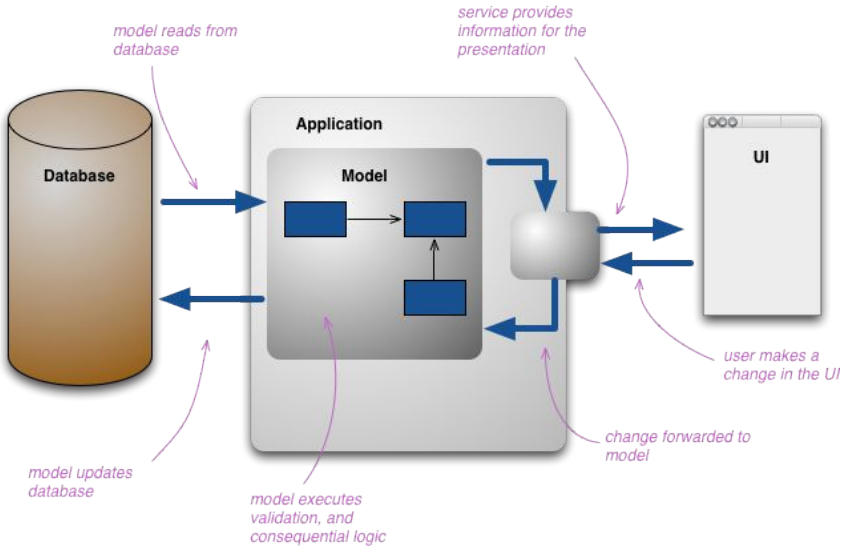
- Many principles can be appied at other levels than the class level
- Example of applying SRP on a lower level: method
- CQS: Command Query Separation (Bertrand Meyer)
  - A method that retrieves data (query) should not adapt anything
    - No side effects: asking a question should not change the answer
  - A method that changes something (command) should not return data
    - Example: an update does not return the changed object

Single Responsibility Principle / CQRS

- Example of applying SRP on a higher level: architecture
- Command Query Responsibility Segraration (CQRS)
  - The model used to manage data does not need to be the same as the one for reporting
    - Other combinations of data
    - Reading and writing have different characteristics: Performance optimisation, consistency…
  - Note: this pattern is only used by some specific systems

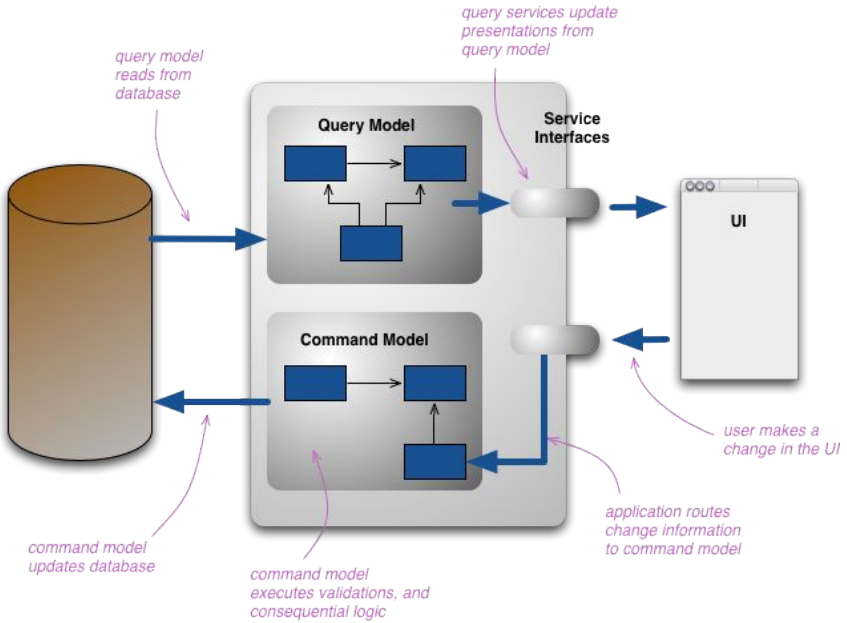
Single Responsibility Principle / CQRS

- Non CQRS
  - Classic SQL database
    - Updates and queries



Single Responsibility Principle / CQRS

- Command Query Responibility Segraration (CQRS)
  - Different interfaces for command and query
  - examples
    - Business Intelligence (BI) often uses a separate database
    - distributed databases
      - query model data can be ot of sync with command model

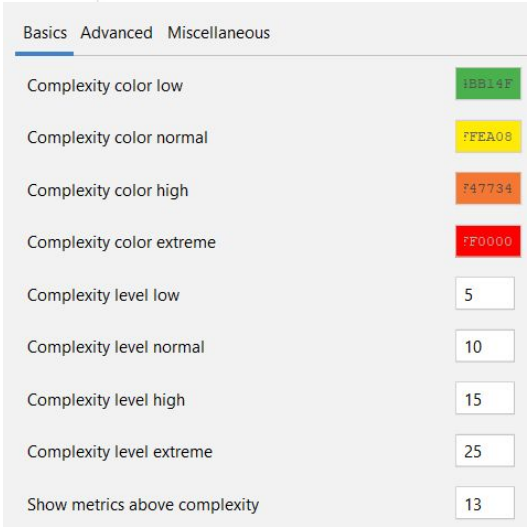


Related

- Cyclomatic complexity
  - # paths through code
  - More sophisticated variant on method length
  - codemetrics plugin for intellij

```
private static double distance(double lat1, double lon1, double lat2, double lon2, String unit) {
  if ((lat1 == lat2) && (lon1 == lon2)) {
    return 0;
  } else {
    double theta = lon1 - lon2;
    double dist = Math.sin(Math.toRadians(lat1)) * Math.sin(Math.toRadians(lat2)) +
        Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) *
            Math.cos(Math.toRadians(theta));
    dist = Math.acos(dist);
```

Complexity is 14 : medium



Open Closed Principle

Open Closed Principle (OCP)

Software must be open for extension and closed for change
(Bertrand Meyer)

- You must be able to extend software (class, mehod) by adding code
- You do not need (to change) existing code

Open Closed Principle (OCP)

Software must be open for extension and closed for change

Avoid changing parts of classes on which other code depends

- If you don't, you need to adapt the code that uses the class as well
  - E.g. don't change the parameters of a public method just like that
- Related Principle: encapsulation

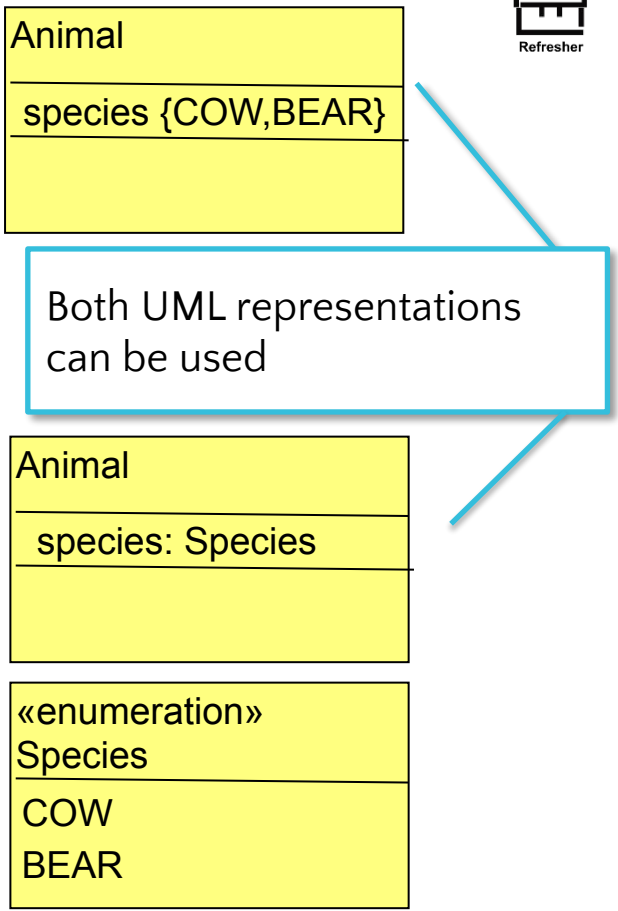
Open Closed Principle

examples/openclosed/ex_0start/Animal

```
public class Animal {
 private Species species;
}
```

examples/openclosed/ex_0start/Species

```
public enum Species {
 COW,
 BEAR
}
```



Refresher

Voorbeeld

https://gitlab.com/kdg-ti/software-engineering-2/examples/solid

Open Closed Principle

Refresher

examples/openclosed/ex_0start/Animal

```
public class Animal {
 String species;

 public boolean sleeps() {
   ZooTime date=ZooTime.now();
   switch (species) {
     case "cow":
       return date.isNight();
     case "bear":
       return date.isWinter();
     default:
       return false;
   }
  }
}
```

examples/openclosed/ex_0start/Zoo

```
args) {
 for (Animal a : animals){

System.out.println(a.getSpecies()
+(a.sleeps()
        ?" sleeps."
         :" is awake.") );
 }
}
// COW is awake.
// BEAR is awake.
```

Karel de Grote Hogeschool

Voorbeeld

https://gitlab.com/kdg-ti/software-engineering-2/examples/solid

Open Closed Principle

Refresher

examples/openclosed/ex_0start

```
public class Animal {
 String species;

 public boolean sleeps() {
   ZooTime date=ZooTime.now();
   switch (species) {
     case "cow":
       return date.isNight();
     case "bear":
       return date.isWinter();
     default:
       return false;
   }
 }
}
```

New!

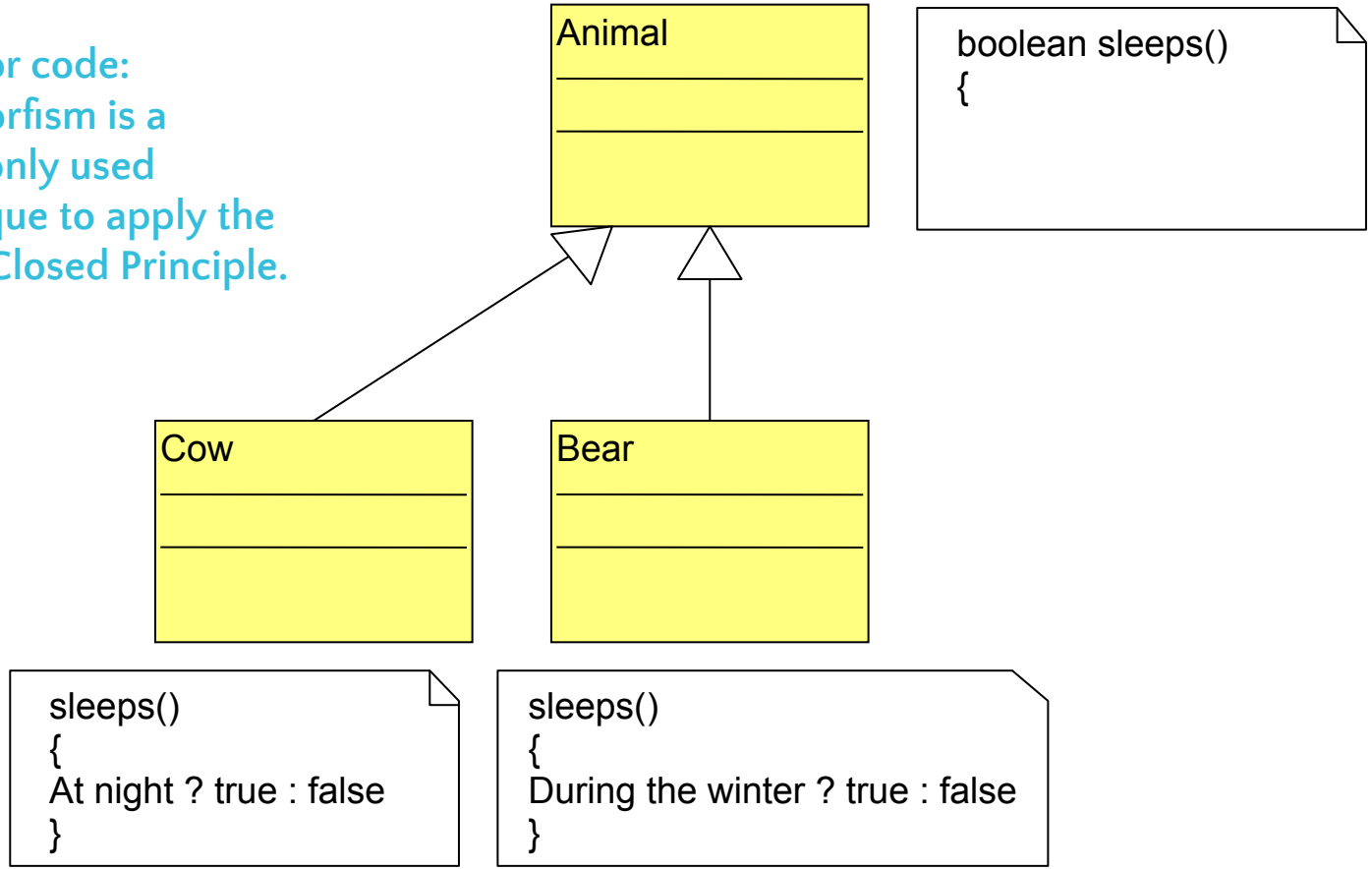
Karel de Grote Hogeschool

Voorbeeld

https://gitlab.com/kdg-ti/software-engineering-2/examples/solid

Open Closed Principle

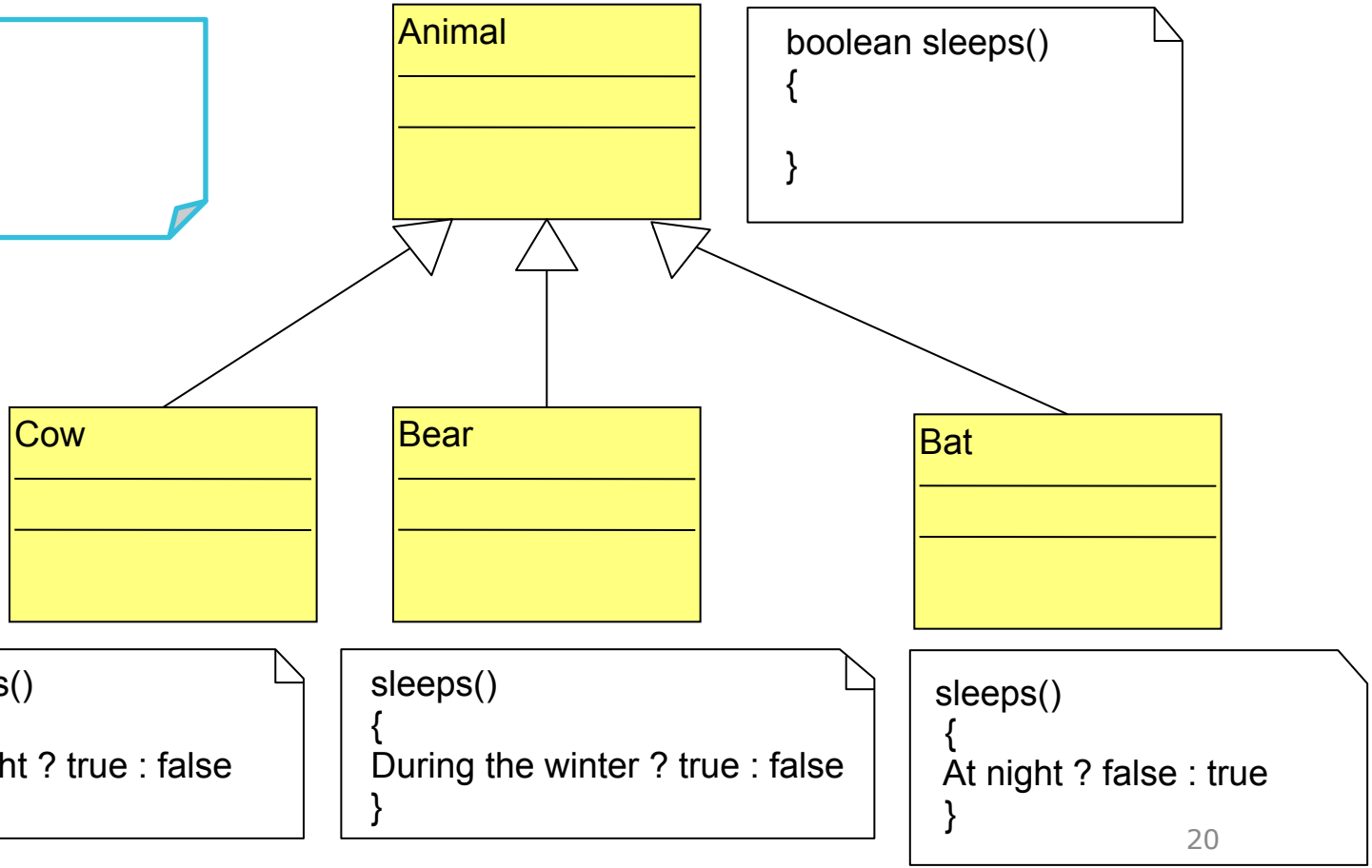
1. Refactor code: polymorfism is a commonly used technique to apply the Open/Closed Principle.



Open Closed Principle

Refresher

2. Extend code



Open Closed Principle

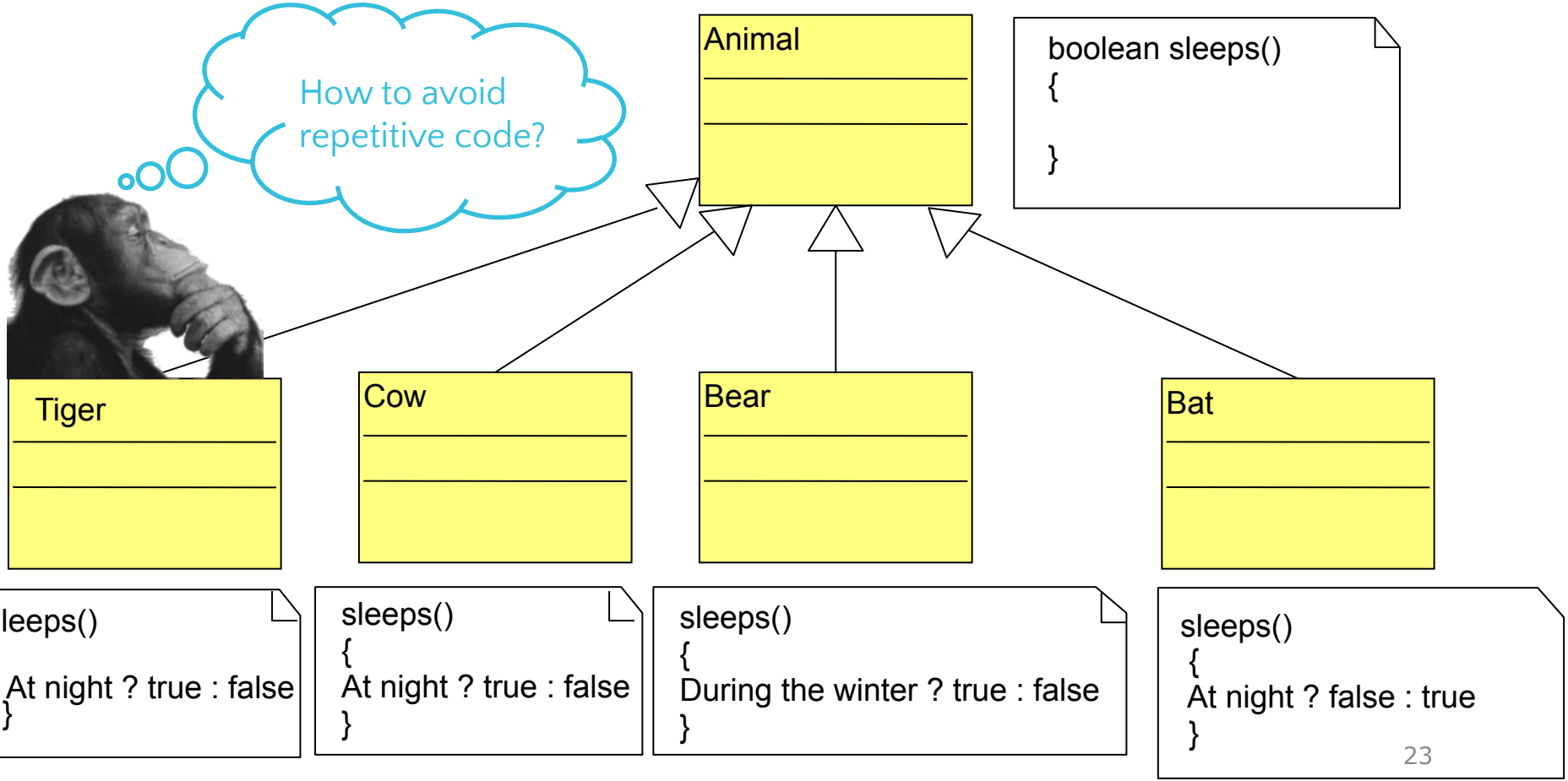
- Because of polymorphism we were able to add a new Species (Bat) without affecting ANY existing code
- Counter guideline: KISS
  - If behaviour does NOT depend on species, the first implementation (using enumeration) is adequate and simpler

Open Closed Principle

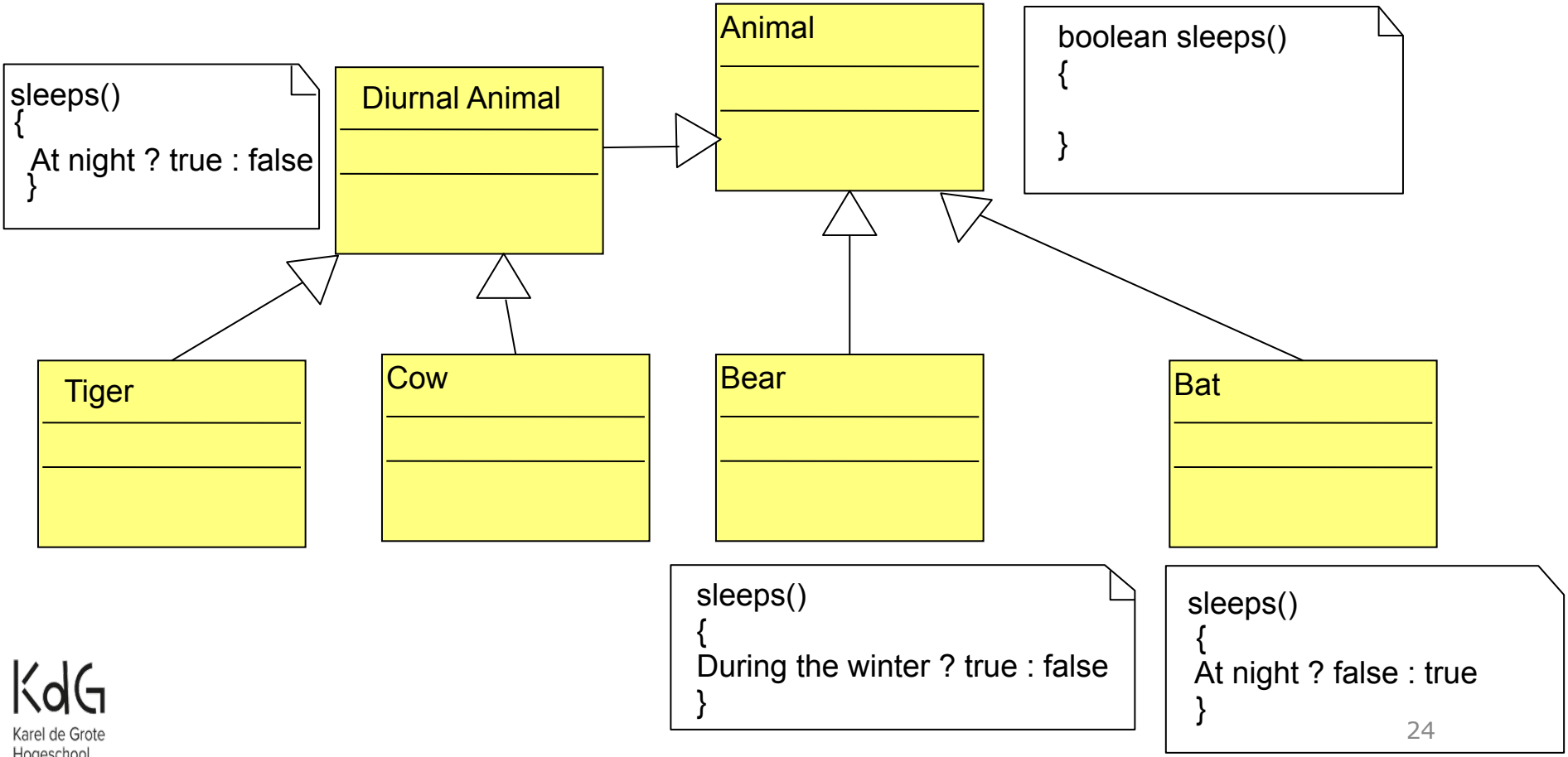
New!

KdG
Karel de Grote Hogeschool

Open Closed Principle



Open Closed Principle



Open Closed Principle

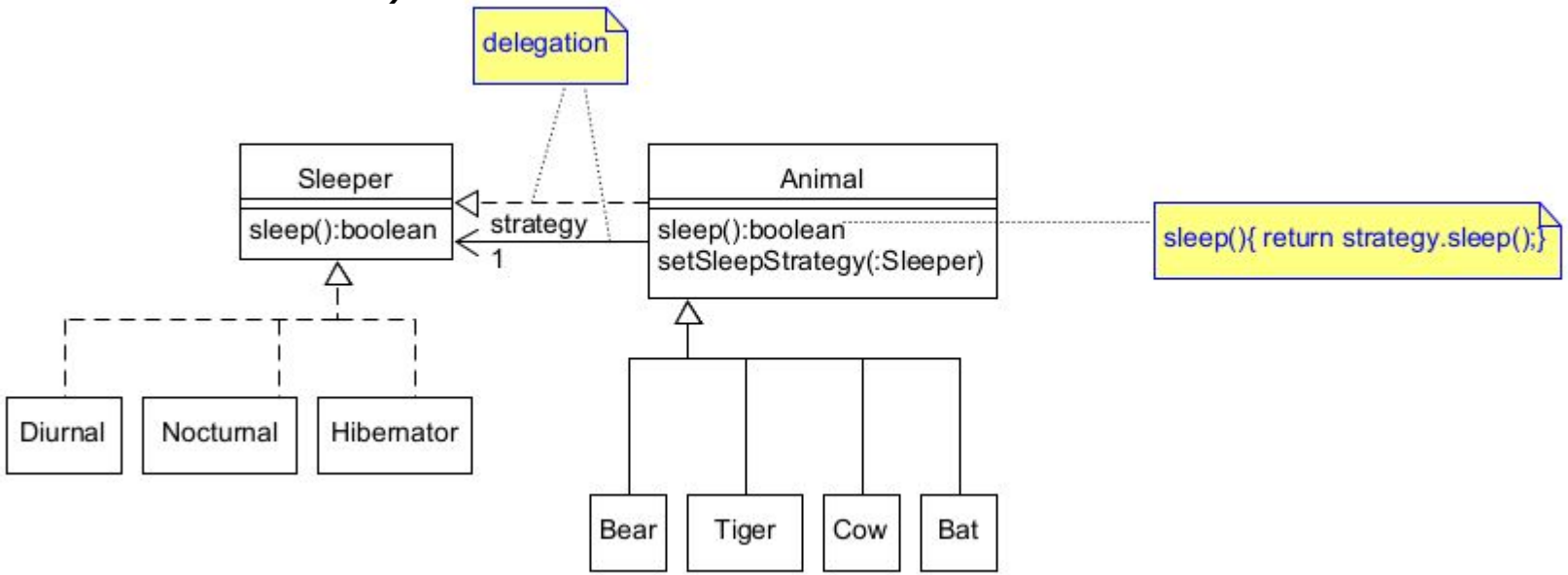
- A polymorfic method must be overridable by its subclasses
  - No stricter access than protected
- Polymorfisme is a GRASP Principle as well

Related: favour composition over inheritance

- Is the hierarchy Animal <- Diurnal Animal <- Cow ideal?
  - Single inheritance
  - inheritance is a static relation

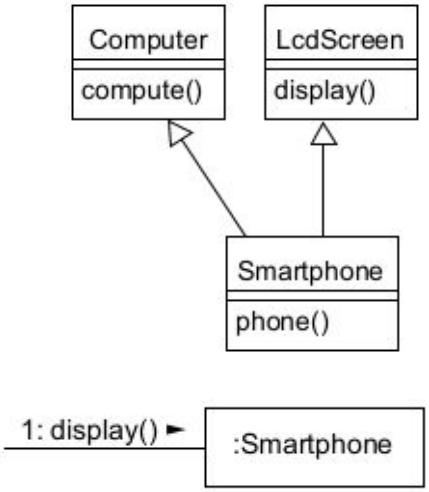
Related: favour composition over inheritance

- Using delegation
- Sleeper conforms to the Single Responsibility Principle
  - A cow can now also be member of another taxonomy (e.g. Animal <- mamal <- Cow)

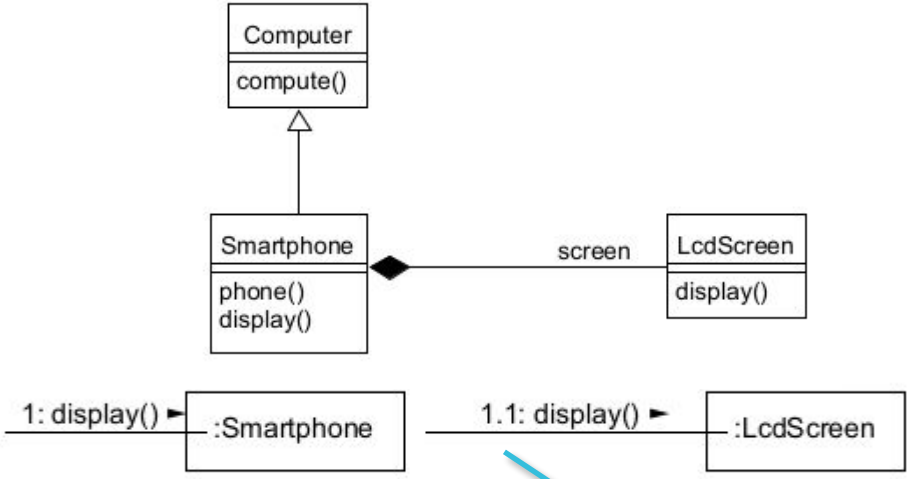


Related: favour composition over inheritance

- Inheritance: smartphone is a kind of LcdScreen

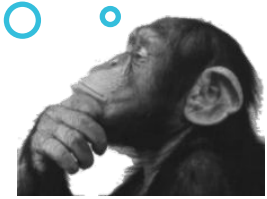


- Compositie: smartphone has an LcdScreen



More code ☹

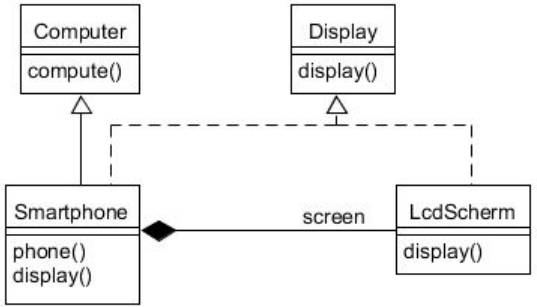
Problem: When using composition, how do I know smartpone has a display method?



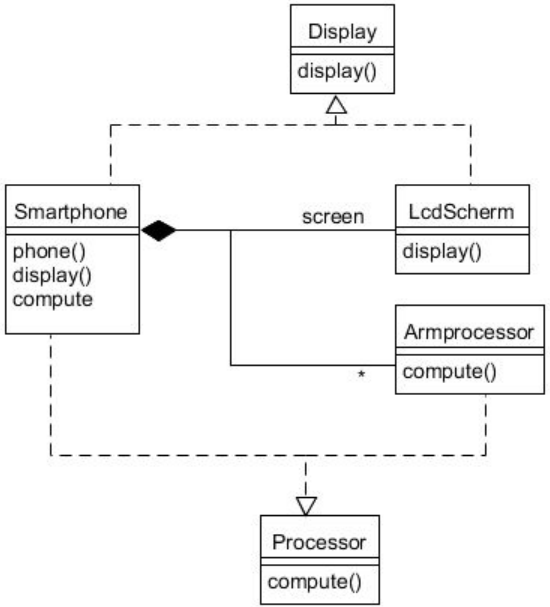
KdG
Karel de Grote Hogeschool

Related: favour composition over inheritance

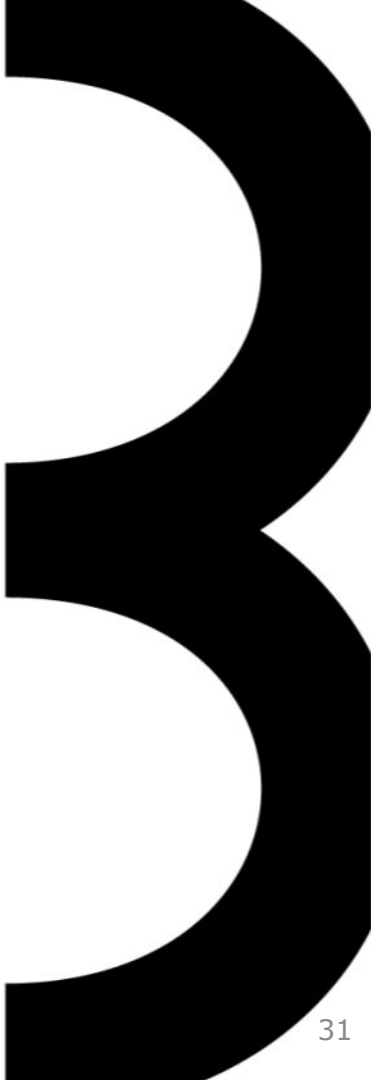
- Solution: Have both classes implement an interface
  - Interface delegation is a composition technique that delegates the methods of a class (Smartphone) to an attribute (screen) implementing the same interface (Display)
- Composition is a flexible relation (inheritance is fixed)
  - You can replace the Smartphone screen
  - You can have multiple screens



Related: favour composition over inheritance



Liskov Substitution Principle



Liskov Substitution Principle (LSP)

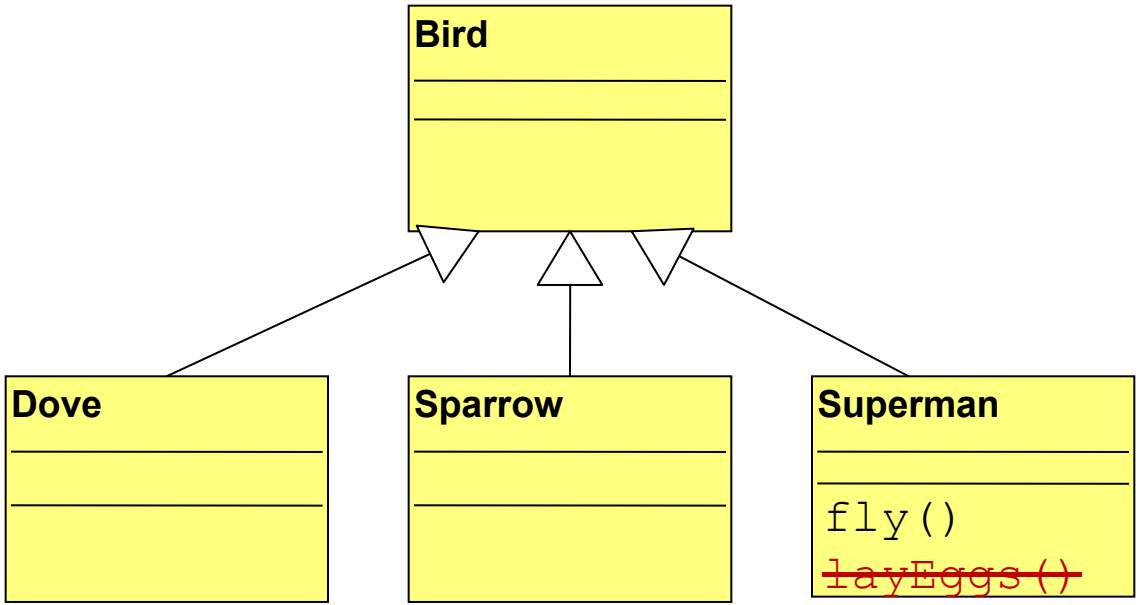
Refresher

An object should always be substitutable by an object of a subclass

- Coined by Barbara Liskov
- The subclass must support all public attributes and methods of the subclass
  - The subclass must fulfill the contract of the superclass
- Inheritance: “is een kind of” relation

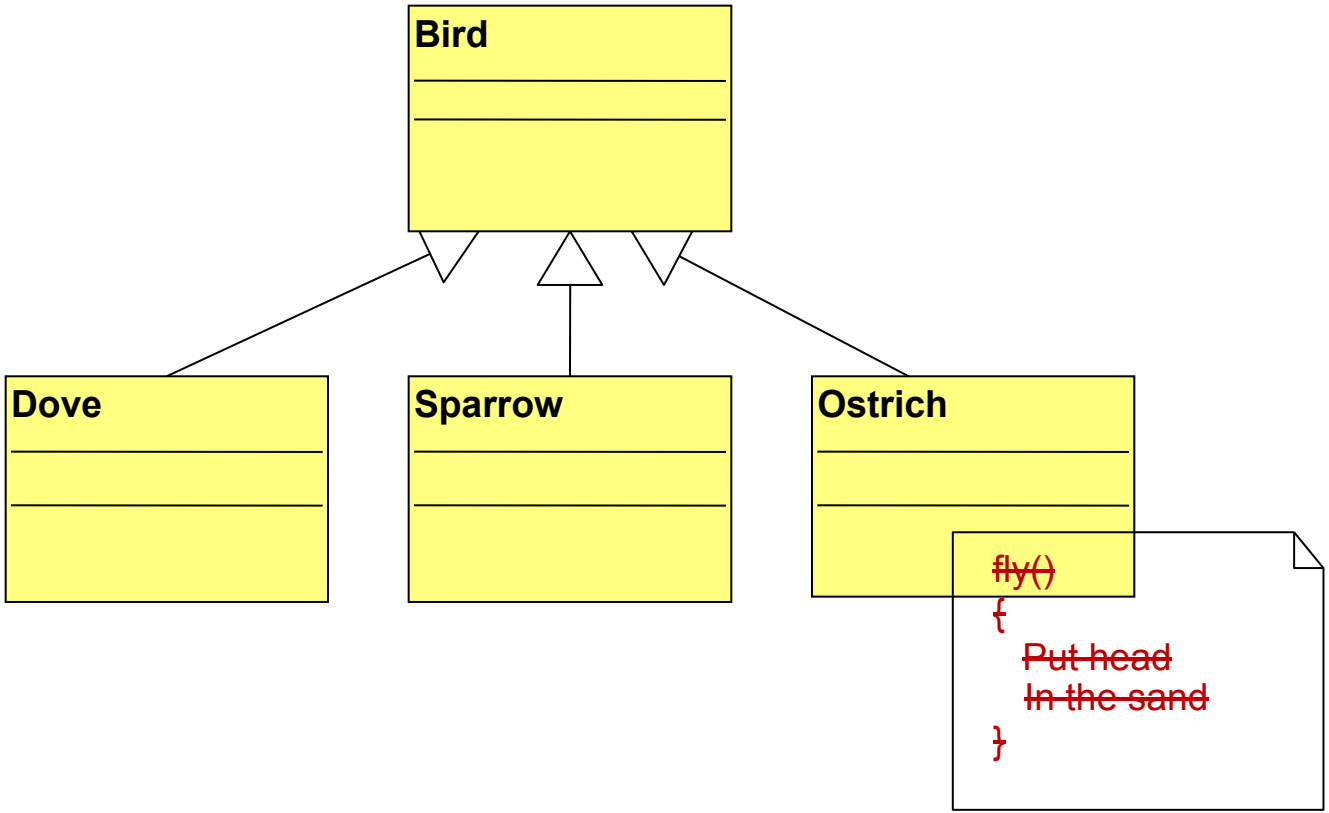
Liskov Substitution Principle

Refresher

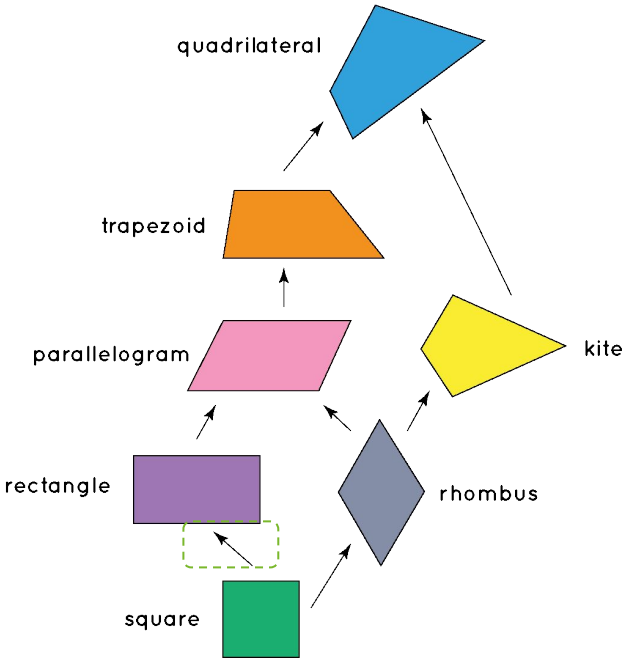


Liskov Substitution Principle

- Methods of subclass must have the same semantics



Liskov Substitution Principle



source: Teacher Benny, 4th grade

```
public class Square extends Rectangle{

  public void setWidth(int width){
    m_width = width;
    m_height = width;
  }

  public void setHeight(int height){
    m_width = height;
    m_height = height;
  }
}
```

Liskov Substitution Principle

```
@Test
rectangleTest() {
   Rectangle r = new Square();
   r.setWidth(5);
   r.setHeight(10);
   AssertEquals(50,r.getArea());
}
```

Honours a different contract (also sets width)

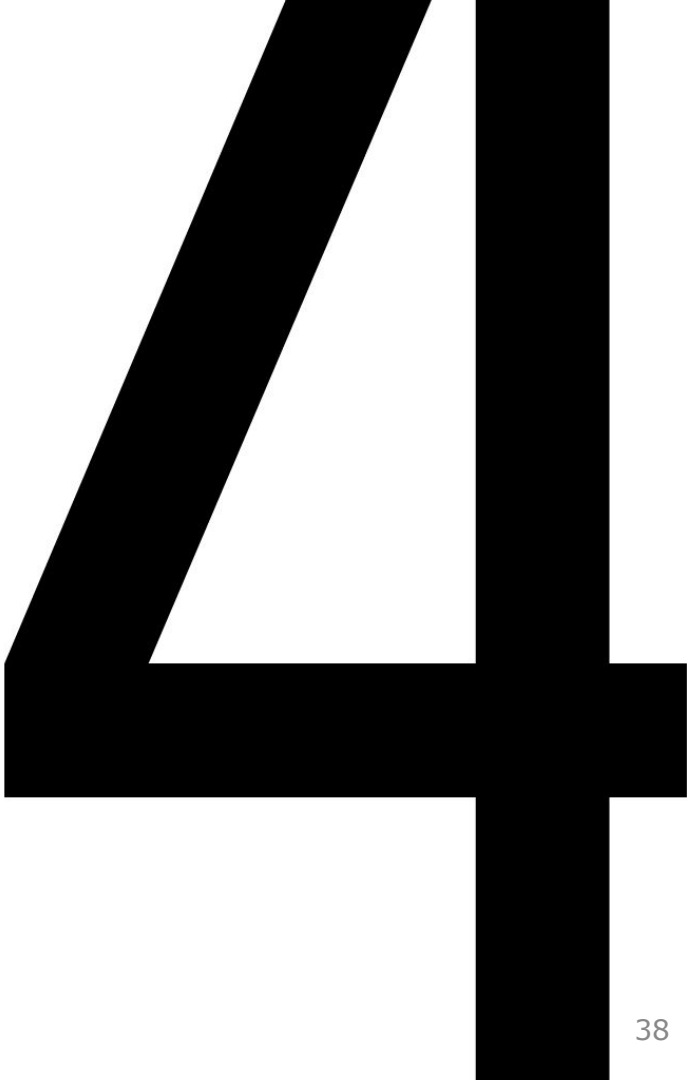
```
AssertionFailedError:
Expected :50
Actual   :100
```

Karel de Grote Hogeschool

LSP Consequences

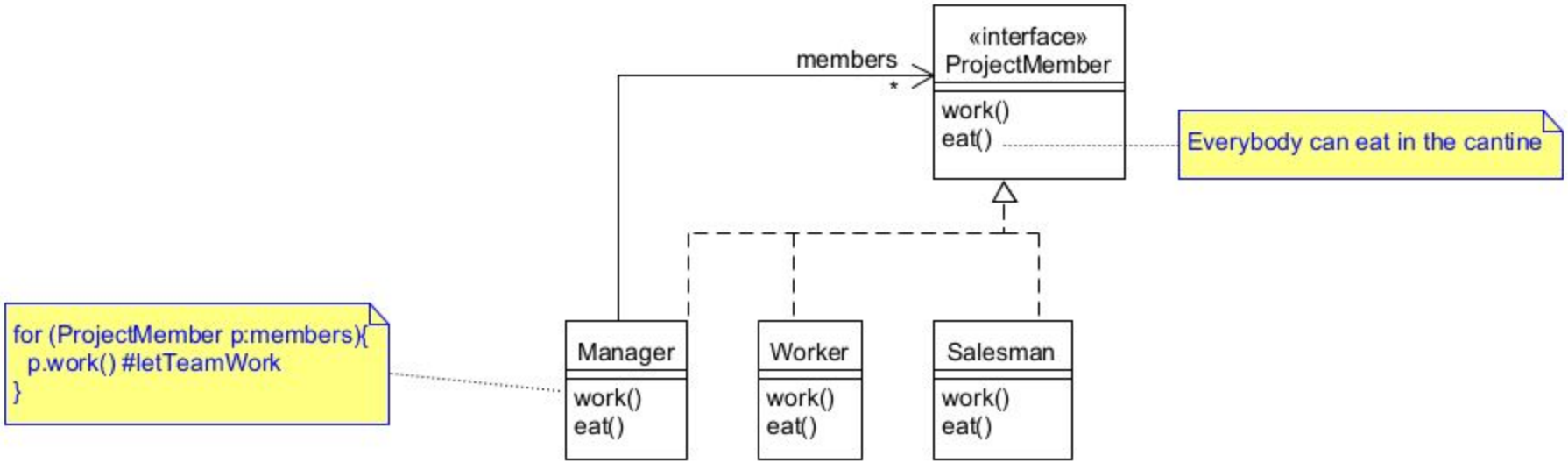
- A subclass cannot have more restrictive methods than the superclass
- A subclass cannot have methods that throw more checked exceptions than the superclass

Interface Segregation Principle

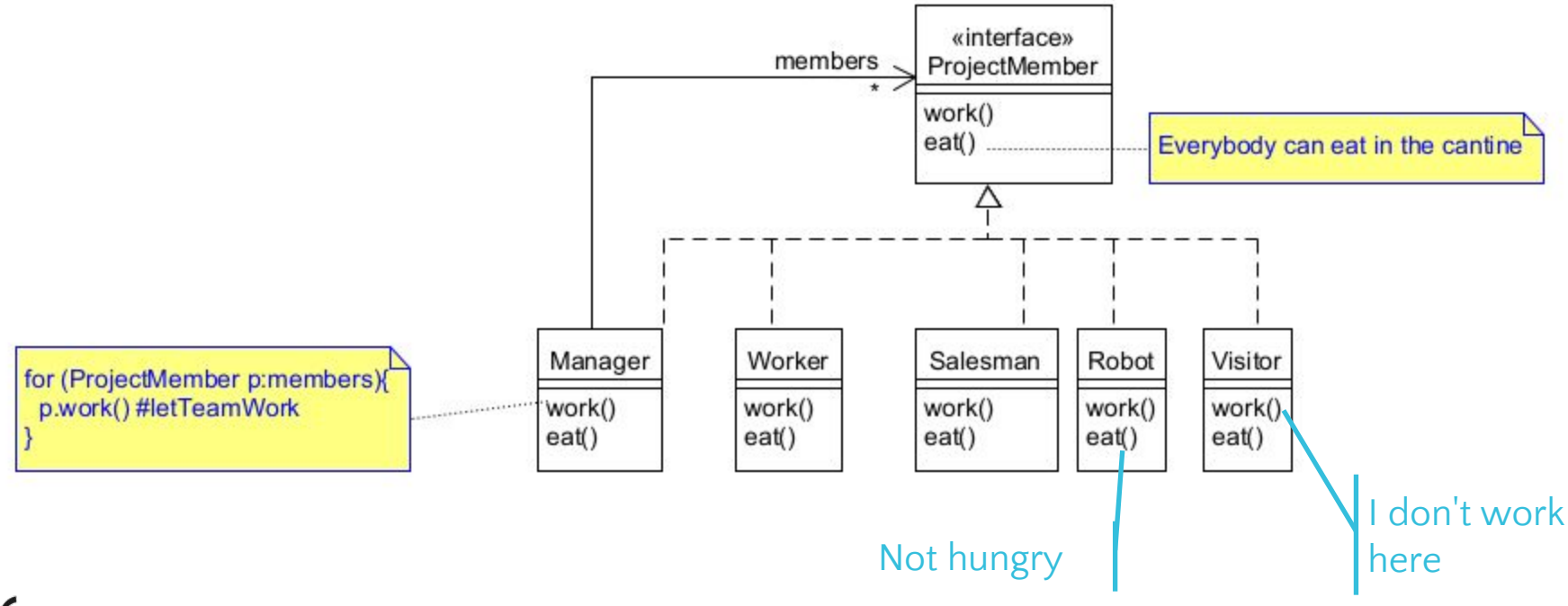


Interface Segregation Principle (ISP)

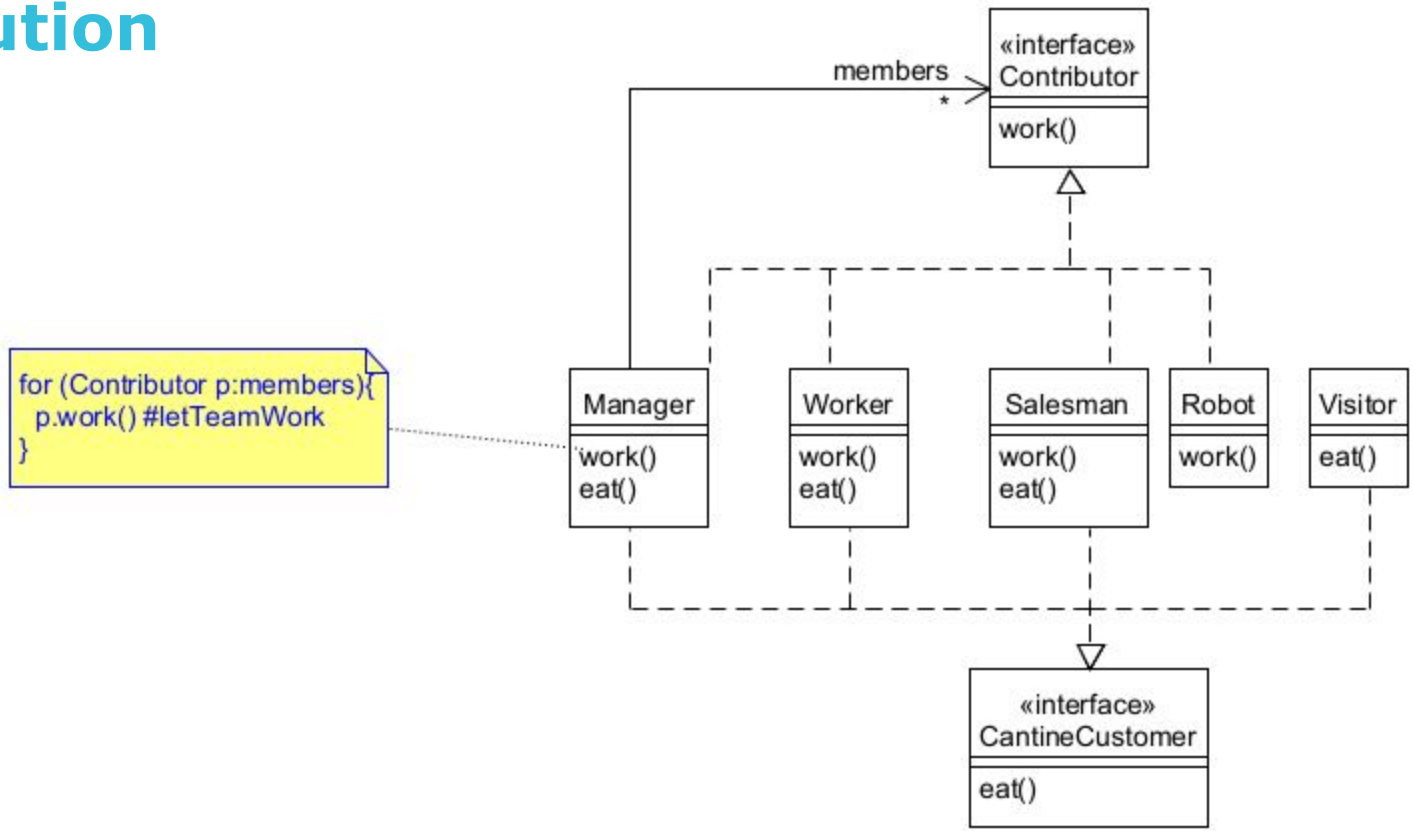
Several specific interfaces are better than one general interface



Interface Segregation Principle (ISP): problem



Interface Segregation Principle (ISP): solution



Interface Segregation Principle (ISP): solution

- Concrete classes can sometimes do a mix of things
- Split them up when defining interfaces for those classes

Interface Segregation Principle (ISP)

- Example from Java API

```
package java.awt.event

public interface MouseListener {

    public void mouseClicked(MouseEvent e);

    public void mouseEntered(MouseEvent e);

    public void mouseExited(MouseEvent e);

    public void mousePressed(MouseEvent e);

    public void mouseReleased(MouseEvent e);

}
```

Interface Segregation Principle

- Problem: when ilmplementing a listener for mouseclicks you also have to implement the other methods
- Solution: extend MouseAdapter

```
java.awt.event.MouseAdapter

public class MouseAdapter implements MouseListener {
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
}
```

Interface Segregation Principle

`java.awt.event.MouseAdapter`

- Problem:
  - single inheritance: if your listener extends MouseAdapter it cannot inherit from another class
- Solution: distribute the methods of MouseListener over multiple interfaces
  - You can combine them flexibly by implementing multiple interfaces

Interface Segregation Principle

- An interface can have multiple methods, but users should not be forced to implement methods they don't use
- An interface with one method is a functional interface. You can use lambda expressions to implement it.
- Related Principles
  - High cohesion
  - Single Responsibility Principle

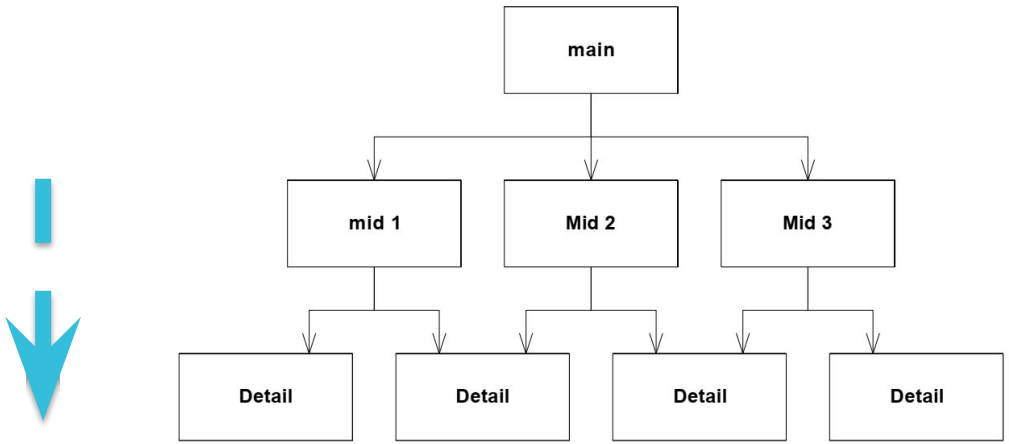
Dependency Inversion Principle

Dependency Inversion Principle (DIP)

Depend on abstractions

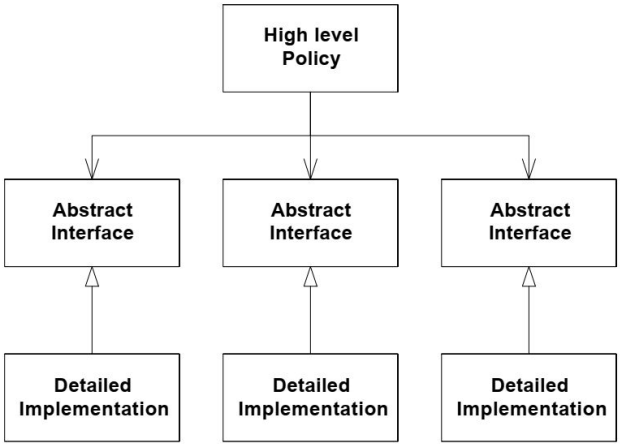
Not on implementations

- A method depends on the methods it calls (procedural style)
- Highest level depends on smallest detail



Dependency Inversion Principle

- Solution: use interfaces. The implementation depends on the interface



Dependency Inversion Principle: problem

```
public class Sale {
  private double total;

  private double getDiscount() {
    double discount = 0.0;
    // Solden
    if( LocalDate.now().getMonth() == Month.JANUARY) {
      discount += total*0.1;
    }
    // big Spender
    if(total >= 300.0){
      discount += 10;
    }
    return discount;
  }
…
}
```

Problem: special offers change all the time

Dependency Inversion Principle: solution

```
public class Sale {
 private List<Discount> promos;

  public double getDiscount() {
    double discount = 0.0;
    for (Discount promo:promos){
      discount += promo.getDiscount();
    }
    return discount;
  }

  public double addDiscount(Discount discount) {
    promos.add(discount);
  }
…
 }
```

Dependency Inversion Principle: oplossing

```
public interface Discount {
 double getDiscount();
}

public class SaleDiscount implements Discount {
 private double total;

 public SaleDiscount(double total) {
  this.total = total;
 }

 @Override
 public double getDiscount() {
  return ( LocalDate.now().getMonth() == Month.JANUARY)?total*0.1:0.0;
 }
}
```

This solution uses the strategy pattern (more detail in a later module)

Dependency Inversion Principle

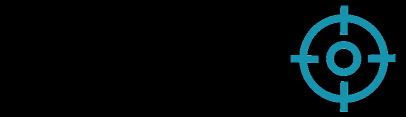
- Use interfaces if a class can have multiple implementations
- Addistional advantages of implementation independence
  - You can write unit tests based on the interface (implementation does not need to exist yet)
  - Mocking: you can supply an implementation that mimics the real class. This allows you to test layers of an application in isolation. Example: you can mock a repository to test without hitting the database

Dependency Inversion Principle

- Related Principles and patters:
  - Dependency Inversion is often used to implement the Open / Closed Principle
  - Low coupling
  - Command pattern: Supply an interface with the command to be executed. Example: `java.awt.event.ActionListener`

```
public interface ActionListener {
    void actionPerformed(ActionEvent e) ;
}
aButton.addActionListener (myActionListener)
```

Samenvatting



S	ingle Responsibility
O	pen / Closed
L	iskov Substitution
I	nterface Segregation
D	ependency Inversion

- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
- https://martinfowler.com/bliki/CQRS.html

KdG Karel de Grote Hogeschool