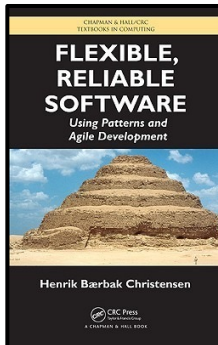
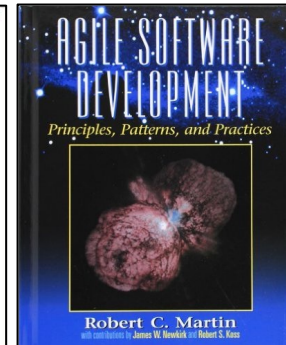
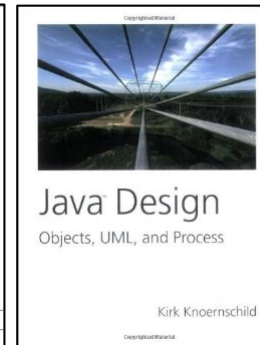
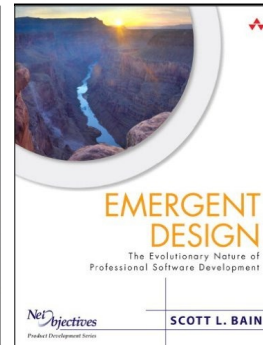
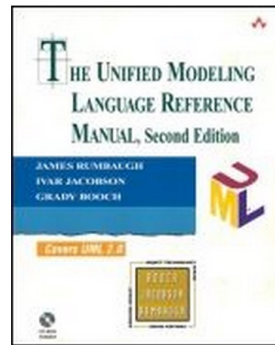
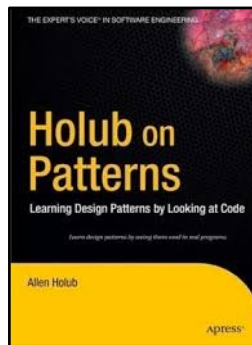
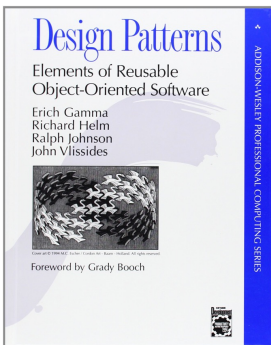
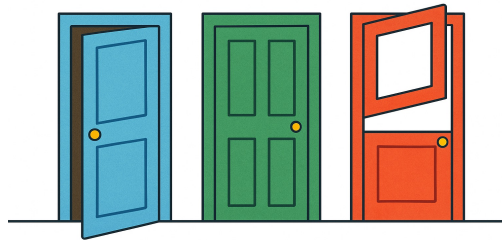


Open-Closed Principle

The Contemporary Version
An Introduction

OPEN FOR EXTENSION,
CLOSED FOR MODIFICATION



deck by



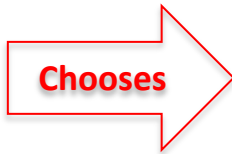
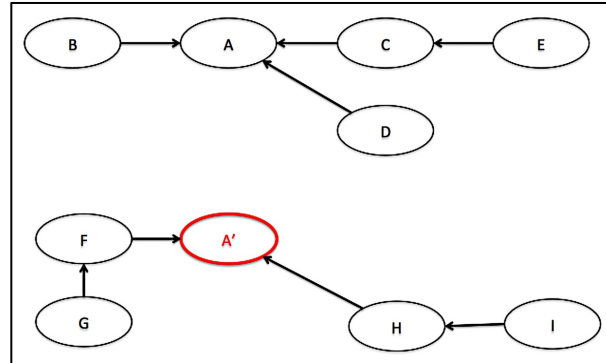
@philip_schwarz



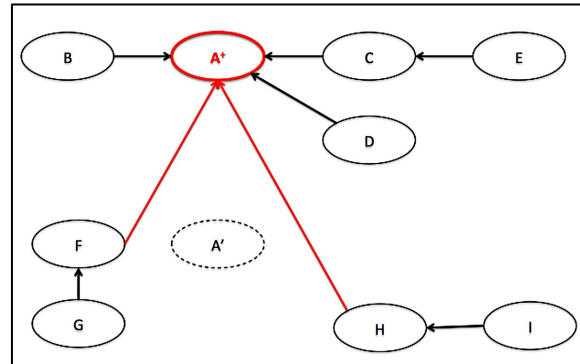
<https://fpilluminated.org/>



**COPY
solution**



**CHANGE
solution**



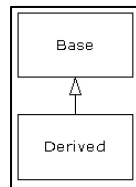
Change by
Modification



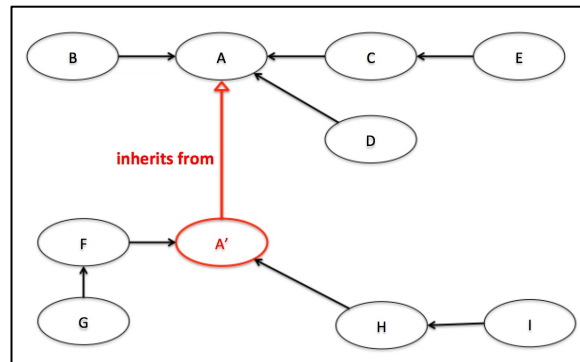
switch



**Organised
Hacking**



**OCP
solution**



Change by
Addition



**But...
extends
is evil!!!!**

But, using **inheritance** is no longer the main approach to satisfying the OCP

`extends` is evil!!!!!!

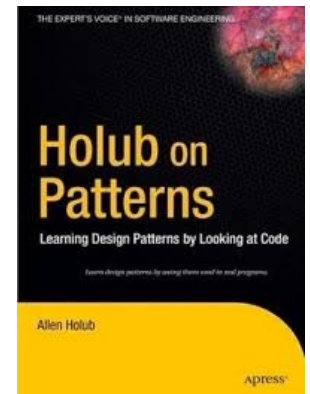


Allen Holub

Why extends is evil



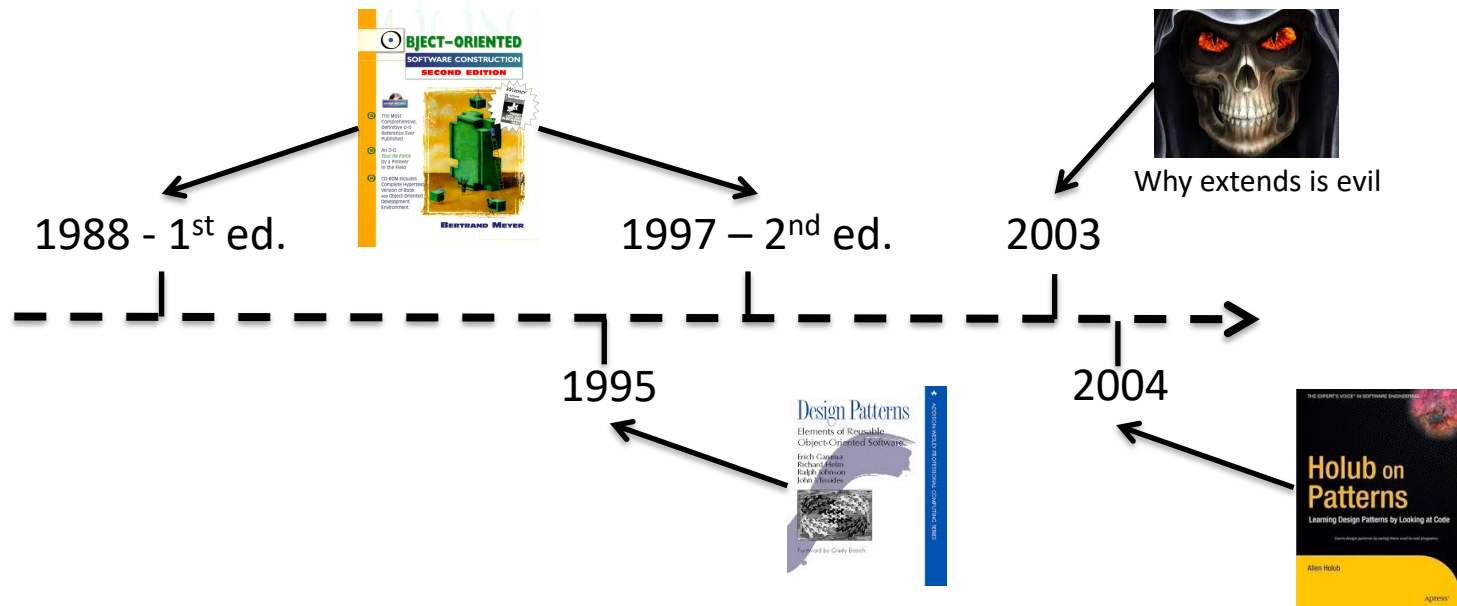
2003

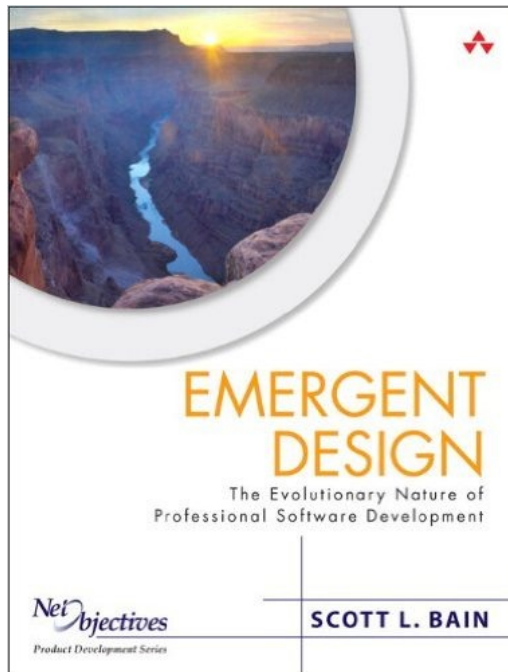


2004

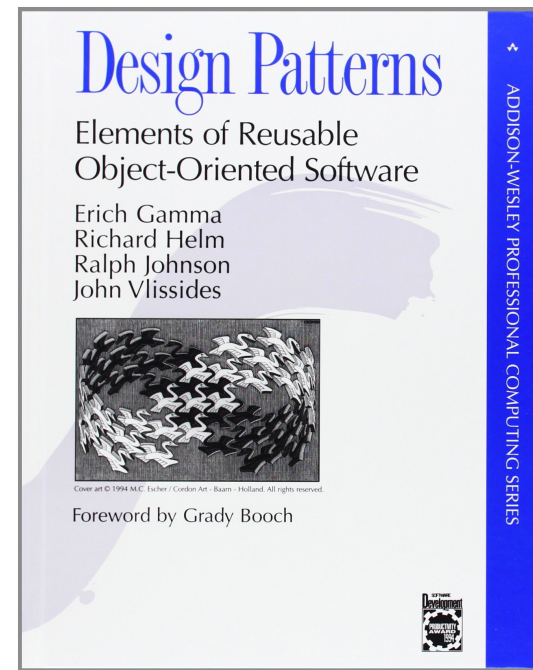
Using inheritance is still one of the ways of satisfying the OCP, and was considered **THE** approach for a long while

That started changing with the emergence of the design techniques presented in **Design Patterns**





2008

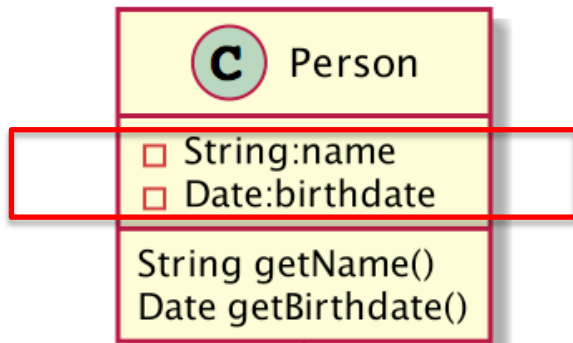


1995

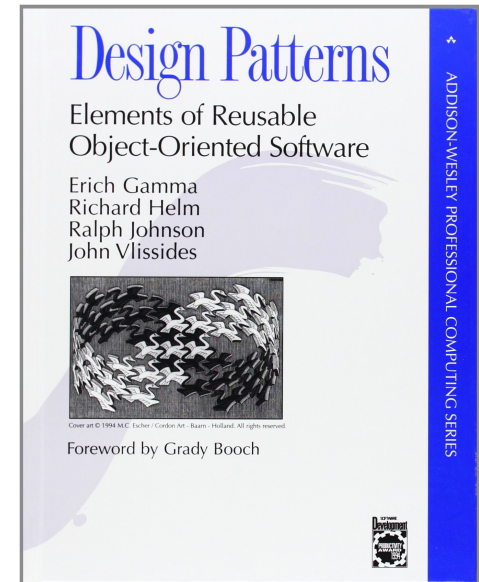
“One major value of studying patterns is that they all, whatever else is true about them, tend to be more **open-closed** than the alternatives”

It's important to understand the difference between an object's **class** and its **type**

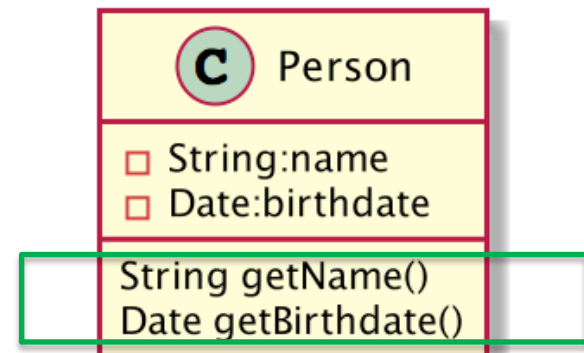
An object's **class** defines how the object is **implemented** (**state** and **operation implementation**)



```
Person(String name, Date birthdate){  
    this.name = name;  
    this.birthdate = birthdate;  
}  
  
String getName(){ return name; }  
  
Date getBirthdate(){ return birthdate; }
```



An object's **type** only refers to its **interface** - the set of requests to which it can respond

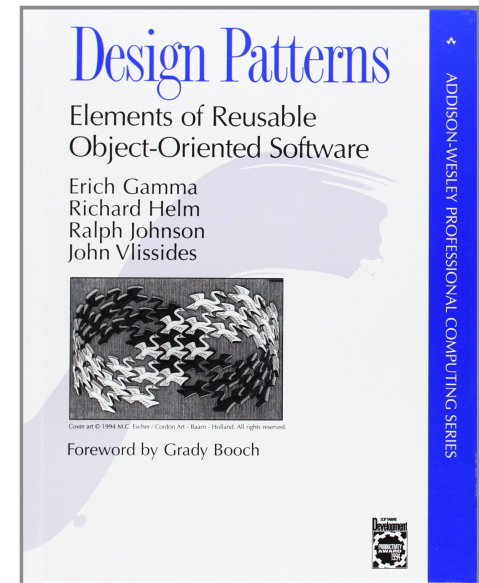
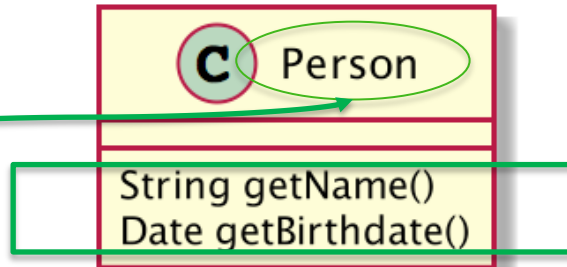


Of course, there is a close relationship between **class** and **type**. Because a **class** defines the operations it can perform, it also defines the object's **type**.

Any **class** C, implicitly forms a **type** C.

```
public class Person
```

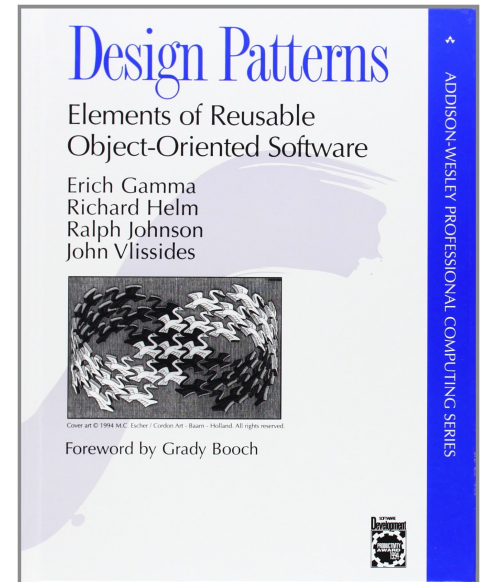
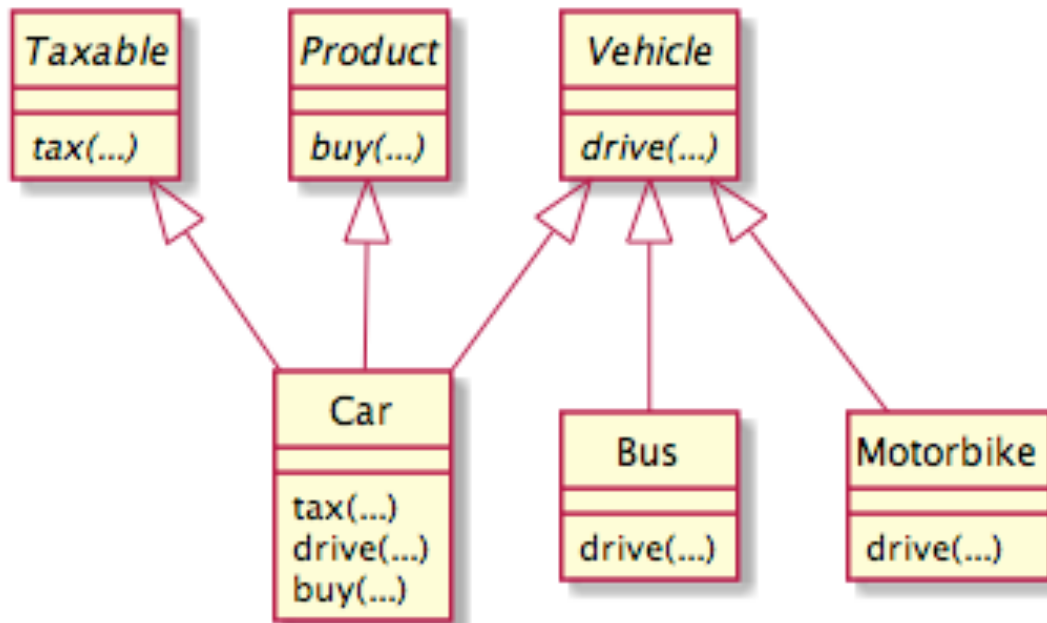
```
{  
    private String name;  
    private Date birthdate;  
    public Person(String name, Date birthdate)  
    {  
        this.name = name;  
        this.birthdate = birthdate  
    }  
    public String getName() {  
        return name;  
    }  
    public String getBirthdate() {  
        return birthdate ;  
    }  
}
```



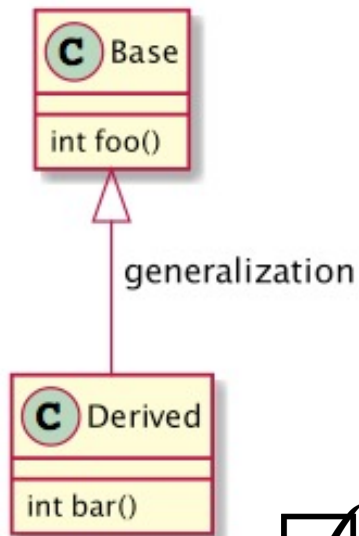
Languages like C++ and Eiffel use **classes** to specify **BOTH** an object's **type** **AND** it's **implementation**.

An object can have **many types**

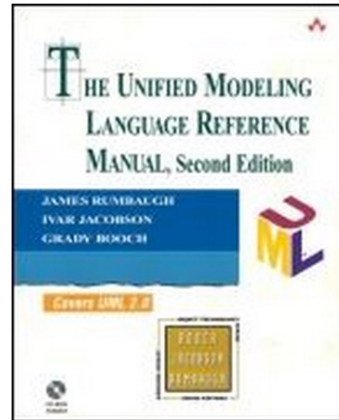
Objects of different classes can have **the same type**



Generalization: a relationship between a more specific and a more general description, used for **inheritance** and **polymorphic** type declarations



Derived inherits **implementation** from **Base**
Derived inherits **type (interface)** from **Base**
 Current and future **specializations** of **Base**
 are **substitutable** for **Base** in clients



S
O
LISKOV
I
D

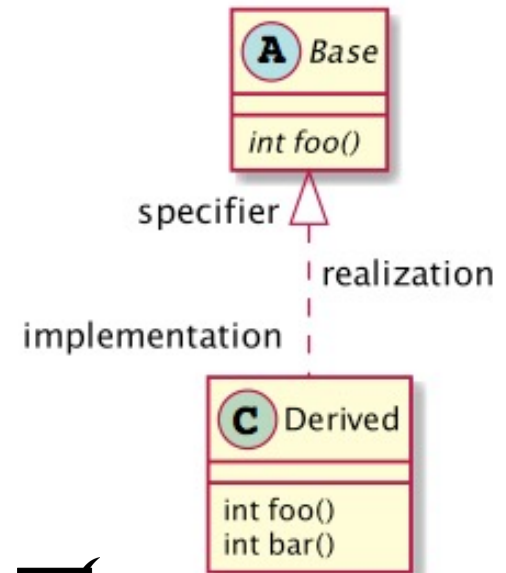


Liskov Substitution Principle
 (Barbara Liskov – 1988)

“[in a Type hierarchy] the supertype’s behavior must be supported by the subtypes: **subtype objects can be substituted for supertype objects** without affecting the behavior of the using code.”



Realization: a relationship between a **specification** and its **implementation**



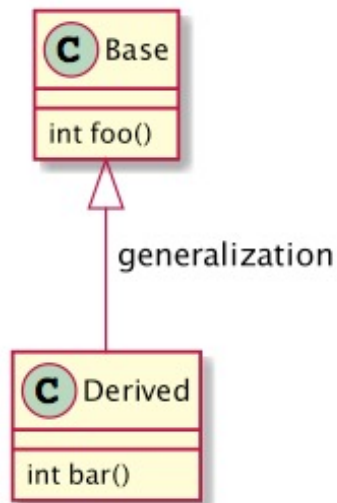
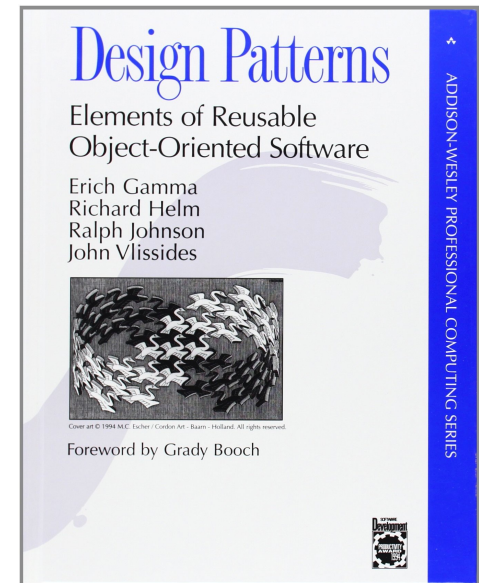
Derived inherits **type** from **Base**

Current and future **realizations** are **substitutable** for **Base** in clients

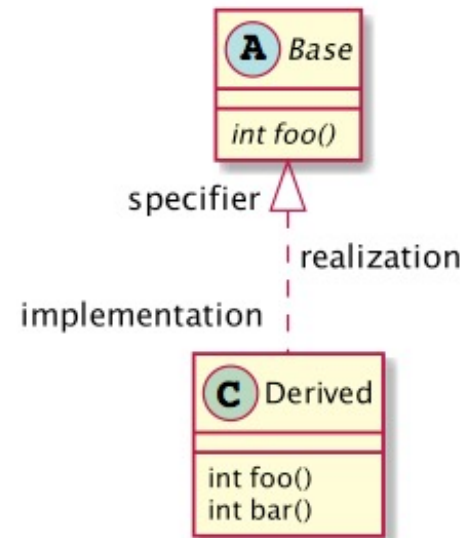
It's also important to understand the difference between **class inheritance** and **interface inheritance** (or **subtyping**)

Class inheritance defines an object's **implementation** in terms of another object's **implementation**. In short, it's a **mechanism for code and representation sharing**

In contrast, **interface inheritance** (or **subtyping**) describes when an object can be used in place of another



Class Inheritance
AND **Interface Inheritance**

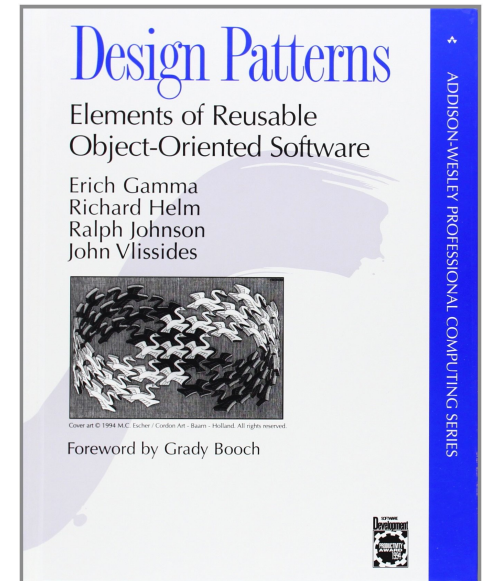


JUST **Interface Inheritance**

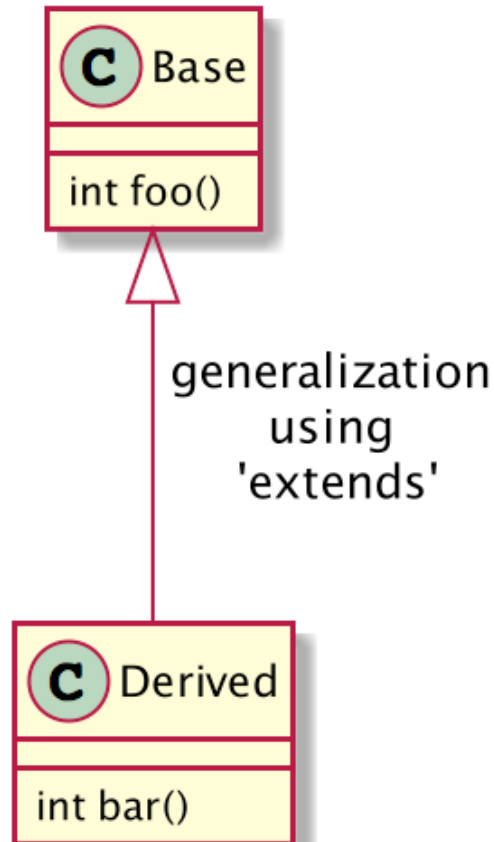
[**interface inheritance** and **implementation inheritance**]

It's **easy to confuse** these two concepts, because **many** languages don't support the distinction between [them]

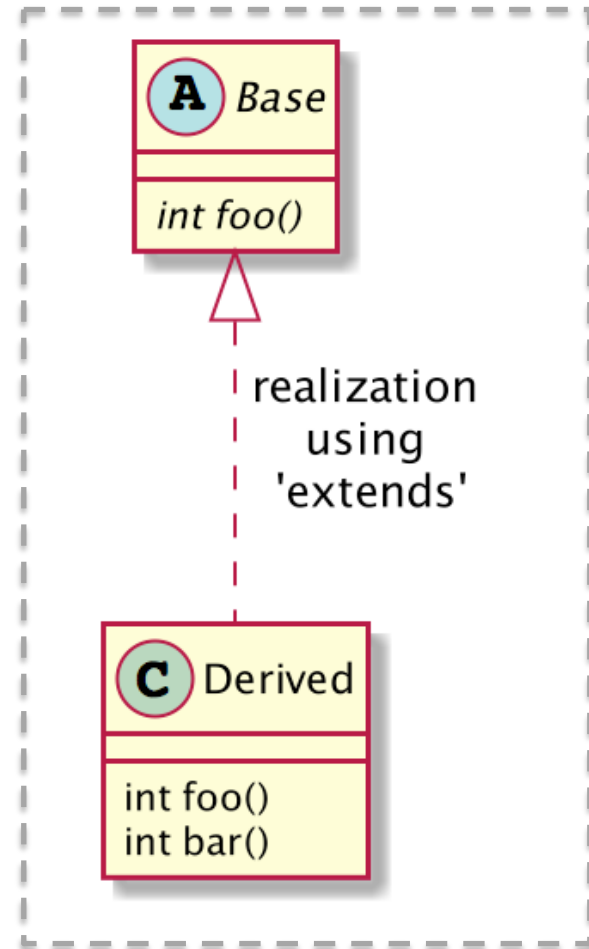
In languages like C++ and Eiffel, **inheritance** means **BOTH interface inheritance** and **implementation inheritance**



If Java were like C++, it would only support realization with **extends**

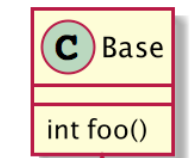


Class Inheritance
AND **Interface Inheritance**

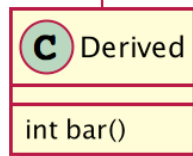


JUST Interface Inheritance

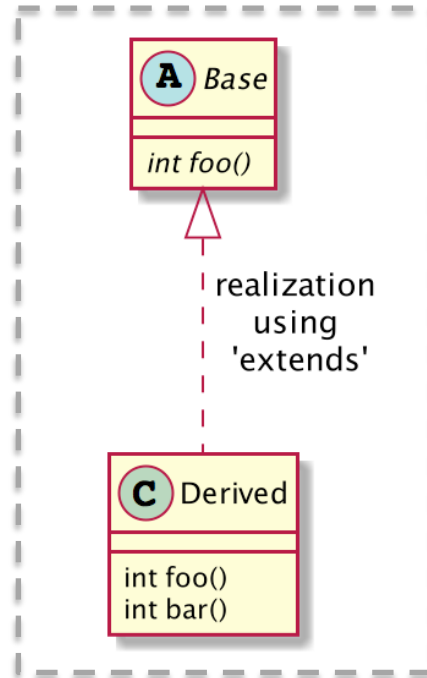
But Java improves on C++ by also supporting realization with **implements**



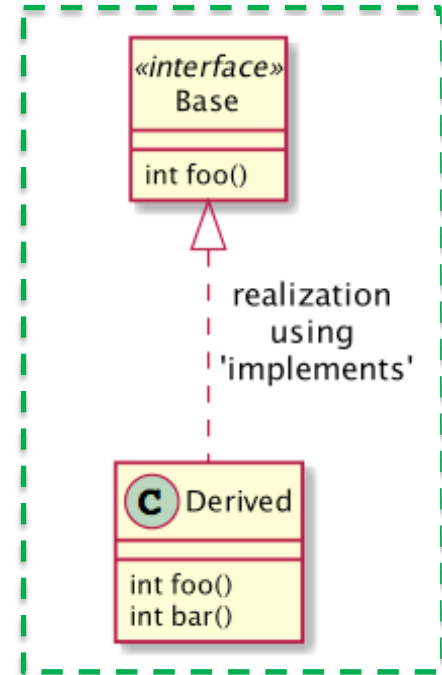
generalization
using
'extends'



Class Inheritance
AND Interface Inheritance



JUST Interface Inheritance



JUST Interface Inheritance



Most statically typed OO languages **conflate** the two concerns of **inheritance** and **subtyping** into a single mechanism. That's a kludge. **Interfaces decouple the two concerns** – Nat Pryce

The **interface** construct is one the few things that **Java really got right** (that and GC) – Steve Freeman





James Gosling was once asked "if you could do java over again, what would you change?"

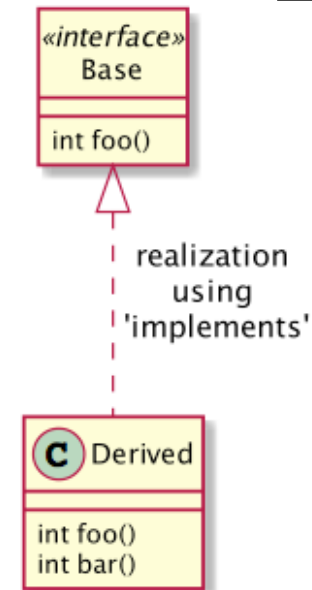
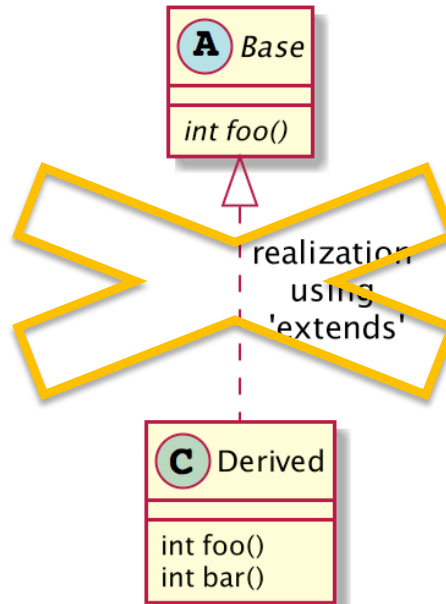
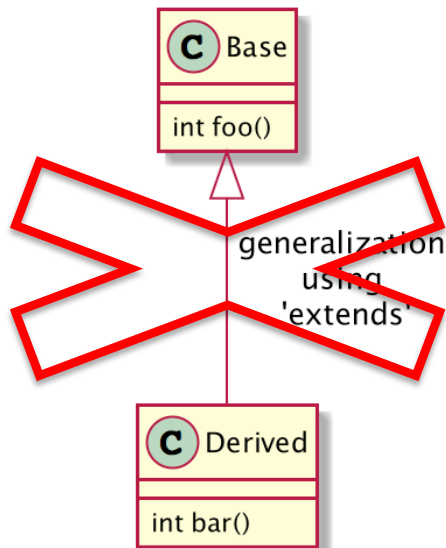
His answer: "I'd leave out classes"



After the laughter died down, he explained that **the real problem wasn't classes per se** but rather **implementation inheritance** (the **extends** relationship).

Interface inheritance (the **implements** relationship) is **much preferred**.

Avoid implementation inheritance whenever possible.



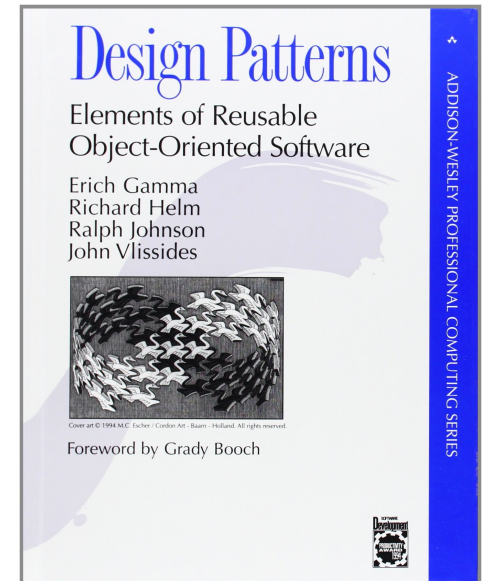
Class Inheritance
AND Interface Inheritance

JUST Interface Inheritance

JUST Interface Inheritance

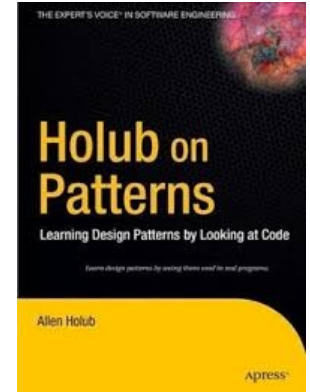
Although most programming languages don't support the distinction between **interface** [inheritance] and **implementation inheritance**, people make the distinction in practice

**Many of the design patterns
Depend on this distinction**





The GoF broke the patterns into **two scopes**

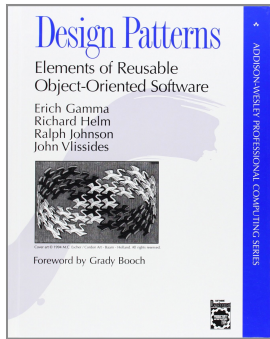


CLASS PATTERNS

require **implementation inheritance** (**extends**) to be reified

OBJECT PATTERNS

should be implemented using nothing but **interface inheritance** (**implements**)



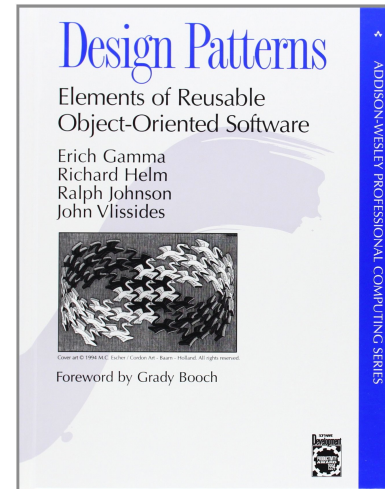
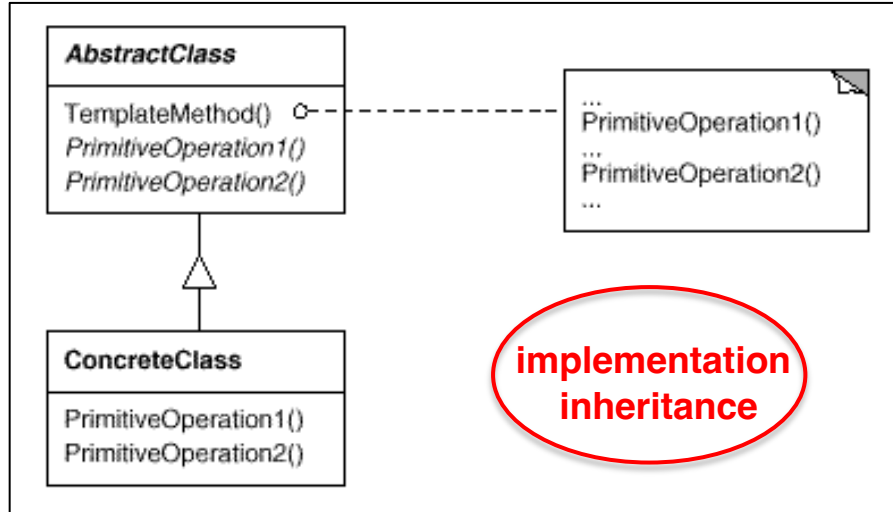
GoF Design Patterns

		Purpose		
		Creational	Structural	Behavioural
Scope	Classes	Factory Method	Adapter(Class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter(Object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

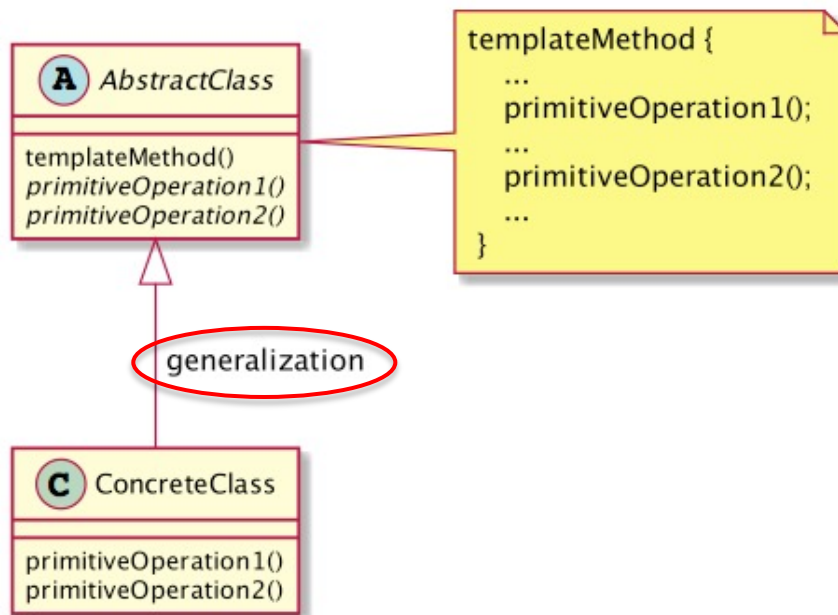
implementation inheritance

interface inheritance

Template Method Pattern

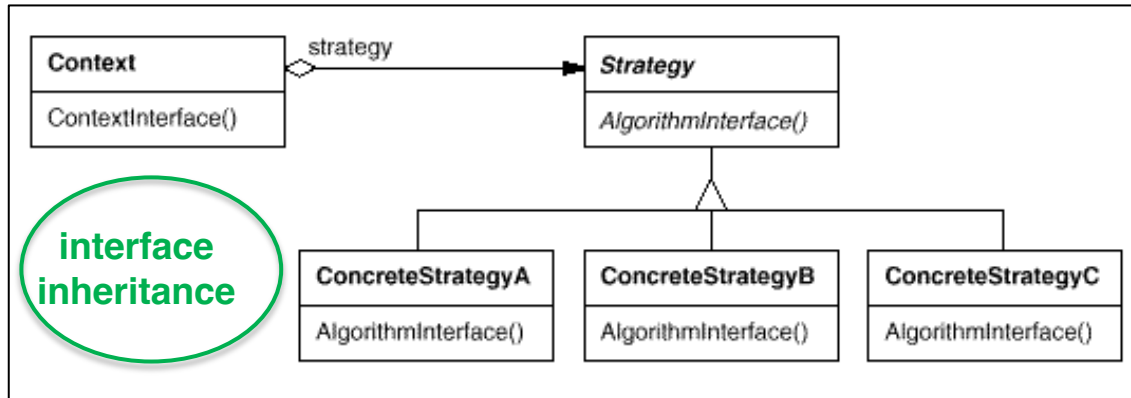


Object
Modeling
Technique

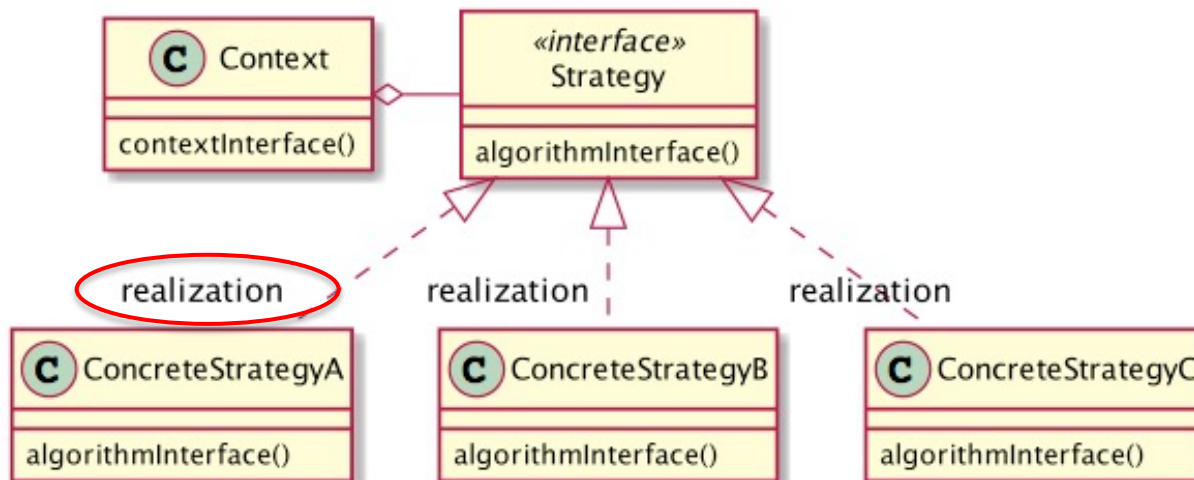
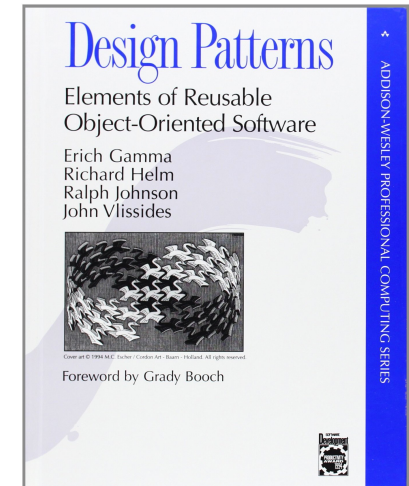


Unified
Modeling
Language

Strategy Pattern



OMT



UML

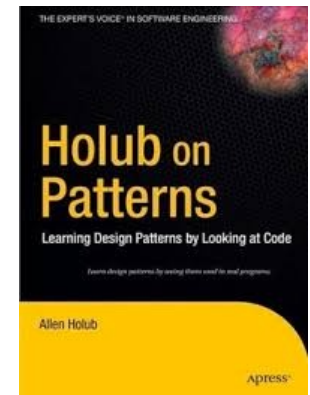


Allen Holub

Purpose		Creational	Structural	Behavioural
c l a s s		Factory Method	Adapter(Class)	Interpreter
				Template Method
S c o o b j e c t		Abstract Factory	Adapter(Object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

implementation inheritance

interface inheritance



2004

It's not an accident that there are many more **Object** patterns than **Class** patterns.

The GoF Design Patterns book is, in fact, largely about replacing **implementation inheritance** (**extends**) with **interface inheritance** (**implements**)



Robert Martin
(Uncle Bob)

The Contemporary Version of the **OCP**

Symptoms of poor design (or **Design Smells**):

Rigidity. The design is difficult to change.

Fragility. The design is easy to break.

Immobility. The design is difficult to reuse.

Viscosity. It is difficult to do the right thing.

Needless complexity. Overdesign.

Needless repetition. Mouse abuse.

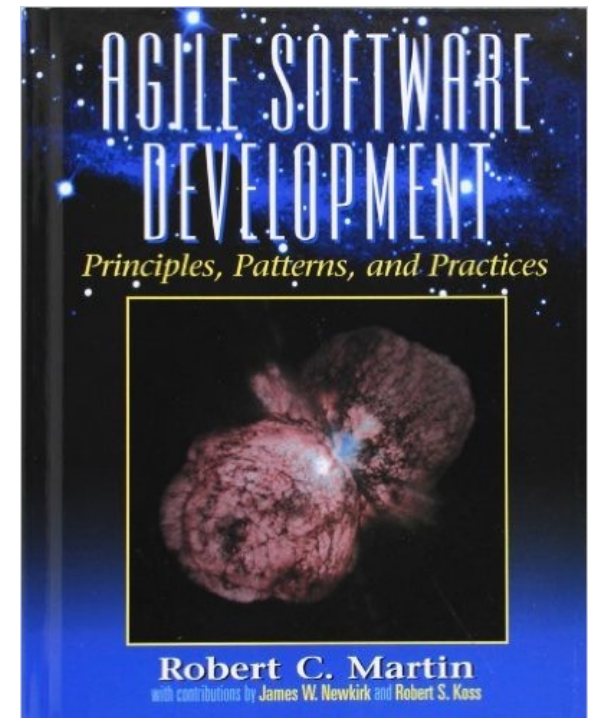
Opacity. Disorganized expression.



Often, the smell is caused by the **violation** of one or more **OO Design Principles**:

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

OO Design Principles help developers eliminate **Design Smells**

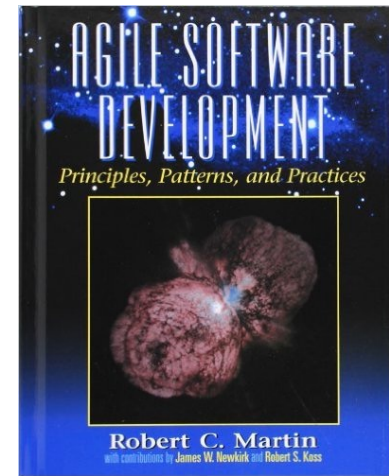


2002



Robert Martin
(Uncle Bob)

The Contemporary Version of the **OCP**



Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

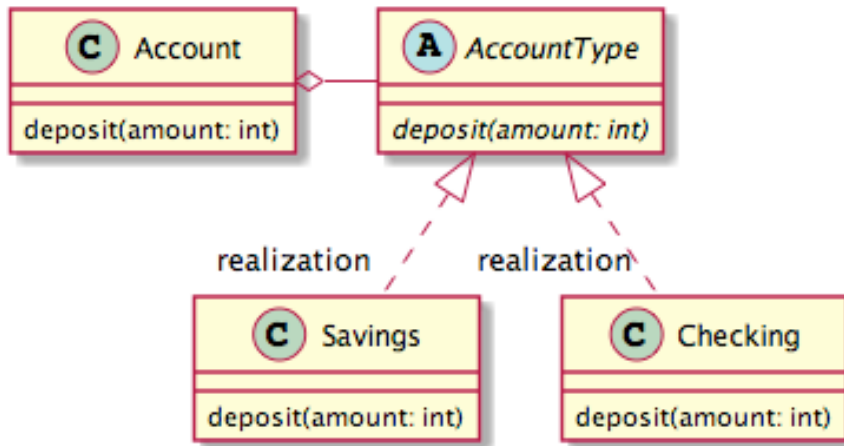
Modules that conform to **OCP** have two primary attributes:

- They are **open for extension**. This means that the behavior of the module can be extended. **As the requirements of the application change**, we can extend the module with **new behaviors** that satisfy those changes. In other words, **we are able to change what the module does**.
- They are **closed for modification**. **Extending the behavior of a module does not result in changes to the source, or binary, code of the module**. The binary executable version of the module...remains untouched.

At the heart of the contemporary **OCP** there is the concept of **abstract coupling**.

The notion that a class is not coupled to another **concrete** class or class that can be **instantiated**. Instead, the class is coupled to other **base**, or **abstract**, classes.

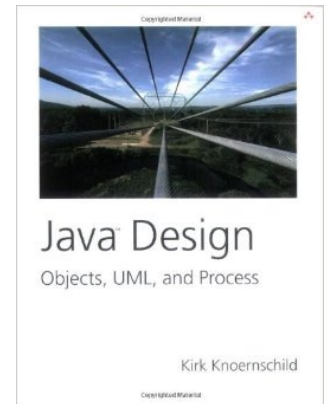
In Java, this **abstract** class can be either a class with the **abstract** modifier or a Java **interface** data type.



We have achieved **OCP** and now can **extend** our system **without modifying its existing code base**.



Kirk
Knoernschild



2001

Account class is
coupled at the abstract level to the
AccountType
inheritance hierarchy

Account isn't directly
coupled to either of the
concrete **Savings** or
Checking classes

So we can **extend** the
AccountType class, creating
a **new class** such as
MoneyMarket, **without**
having to modify our
Account class.

DIP: The Dependency-Inversion Principle



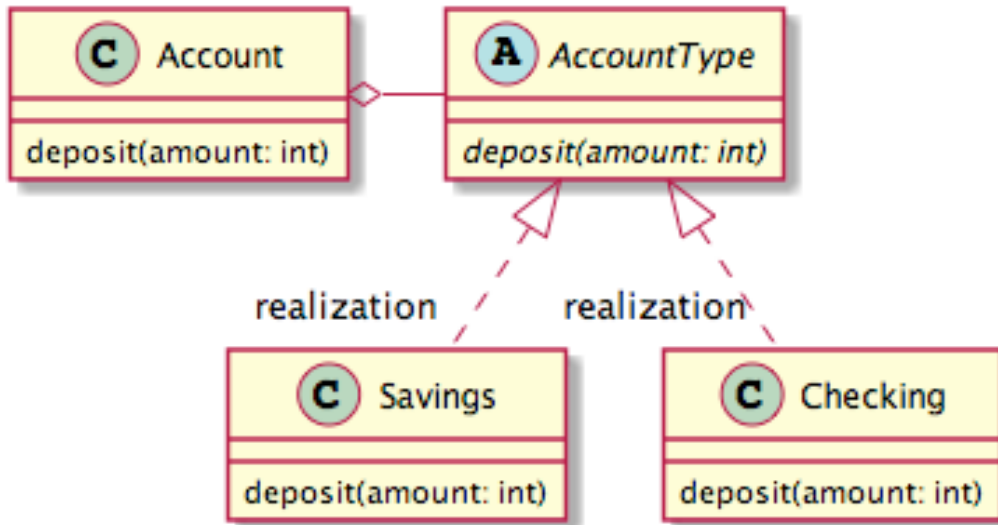
- High-level modules should not **depend** on **low-level** modules. Both should depend on **abstractions**.
- Abstractions should not **depend** on **details**. Details should depend on **abstractions**.

Depend upon **abstractions**. Do not depend upon **concretions**.

DIP formalizes the concept of **abstract coupling** and clearly states that we should couple at the **abstract level**, not at the **concrete level**

- There exists a **striking similarity between DIP and OCP**. In fact, these two principles are closely related
- Fundamentally, **DIP** tells us how we can adhere to **OCP**
- if **OCP** is the desired end, **DIP** is the means through which we achieve that end.





DIP ✓



AccountType is **abstract**, so the coupling of **Account** to **AccountType** is **abstract coupling**, and so is the coupling of **Savings** and **Checking** to **AccountType**



High-level module **Account** does not depend on **low-level** modules **Savings** and **Checking**. **Account**, **Savings** and **Checking**, all depend on an **abstraction**: **AccountType**.



LSP: Subclasses should be substitutable for their base classes

The **LSP** is one of the prime enablers of **OCP**

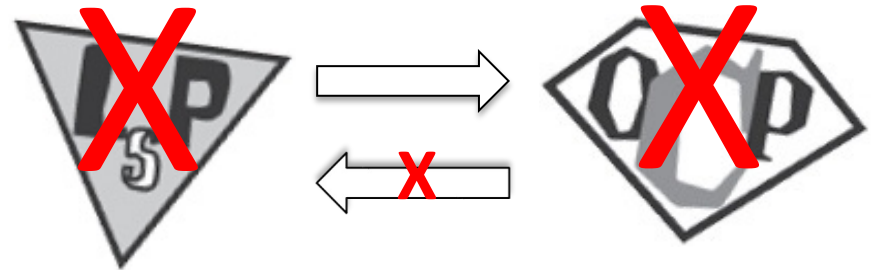
Think of **LSP** as an extension to **OCP**

S
O
LISKOV
I
D

In order to take advantage of LSP,
we must adhere to OCP

because **violations of LSP**
also are violations of OCP

but not vice versa



But why?



often by explicitly
checking the classes
of objects

Change by
Modification

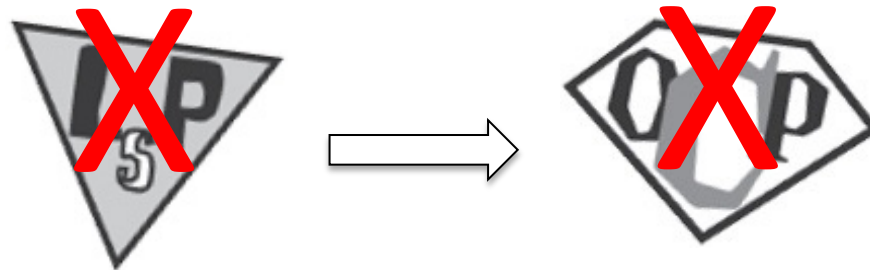
Untrustworthy hierarchies [those violating the LSP] force objects that interact with them to know their **Quirks**

when asked to use one ... [inexperienced developers] will **embed knowledge of its Quirks into their own code**

```
if (bicycle instanceof MountainBike)
{
    ...
}
if (bicycle instanceof MountainBike)
{
    ...
}
if (bicycle instanceof MountainBike)
{
    // code that knows about Quirks
}
```



every violation of the LSP is a latent violation of the OCP



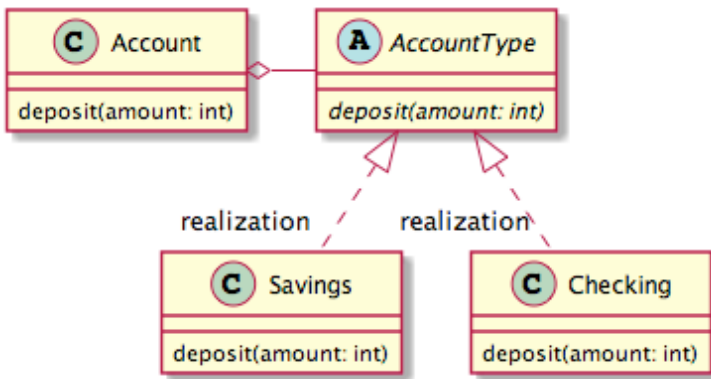
because **in order to repair the damage** ... we are going to have to **add if statements** and **hang dependencies upon subtypes**



S
O
LISKOV
I
D

In its simplest form, **LSP** is difficult to differentiate from **OCP**, but a subtle difference does exist.

OCP is centered around **abstract coupling**. **LSP**, while also heavily dependent on **abstract coupling**, is in addition heavily dependent on **preconditions** and **postconditions**, which is **LSP**'s relation to **Design by Contract**

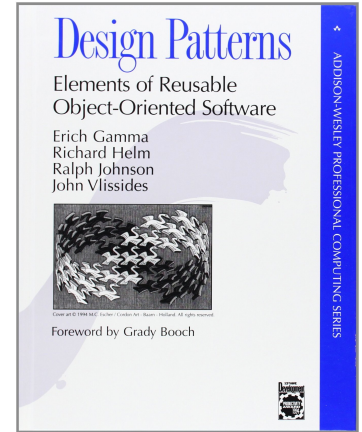


Savings and **Checking** are substitutable for **AccountType**

LSP ✓

Abstract Coupling

“Program to an interface, not an implementation”



Abstract coupling is

the means through which **LSP** achieves its flexibility

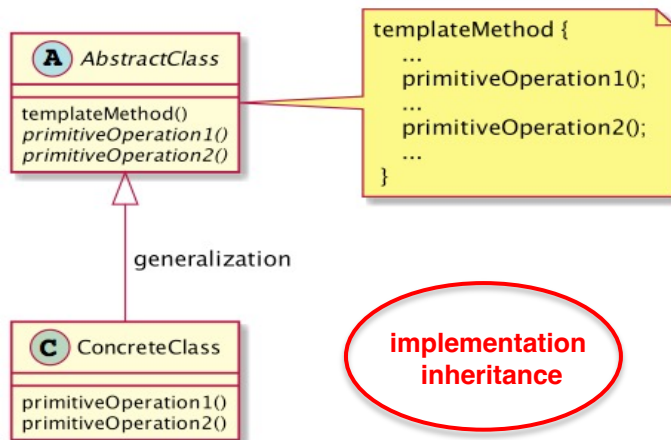
the mechanism required for **DIP**

and the heart of **OCP**

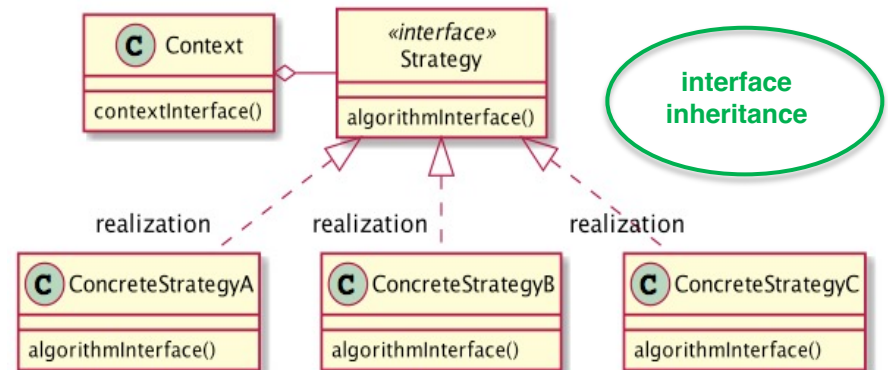


The **Template Method** and **Strategy** patterns are the most common ways of satisfying **OCP**

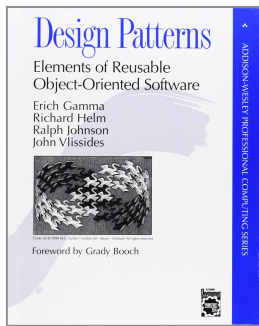
Template Method Pattern



Strategy Pattern



Is one as good as the other? Can they be used interchangeably?



“it is easy to confuse **implementation inheritance** with **interface inheritance** because many languages don’t support the distinction between them”

“Many of the design patterns Depend on this distinction”



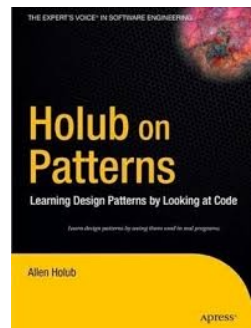
Interface inheritance (the **implements** relationship) is much preferred.

Avoid implementation inheritance whenever possible.



“The GoF Design Patterns book is, in fact, largely about **replacing implementation inheritance (extends) with interface inheritance (implements)**”

“Template Method **has little to recommend it in most situations.** Strategy for example, typically provides **a better alternative.**”





Change by
Modification



Change by
Addition

COPY
solution



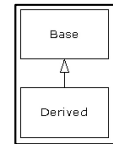
Source code copy
solution

CHANGE
solution



Parametric
solution

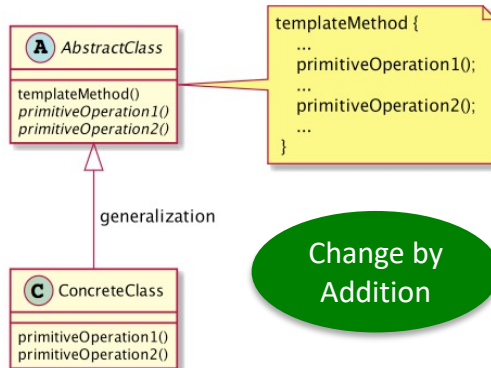
OCP
solution



Organized
Hacking

“Another way of characterizing
Change by Addition that you
may come across is the **Open**
Closed Principle”

Template Method Pattern



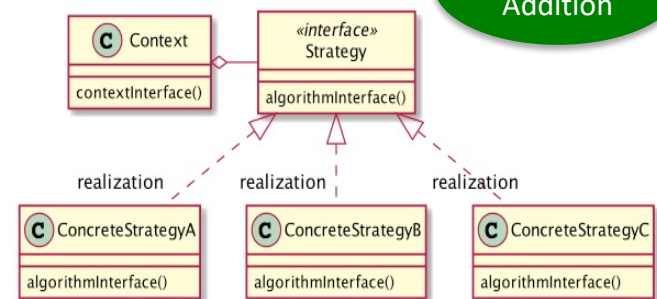
Polymorphic solution

You **encapsulate the variability** points in instance methods. These can then be overridden in subclasses, one for each required variant

“Meyer is generally credited as having originated the term [OCP], however his focus (being in **the golden days of OO inheritance**) was on the **polymorphic approach**”



Strategy Pattern



Compositional solution

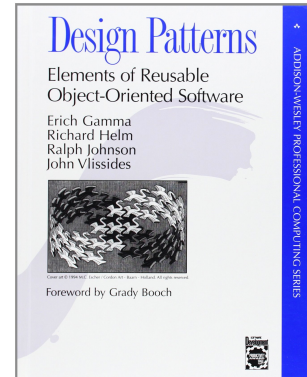
You **encapsulate the variability points** in a well defined interface and use **delegation** to **compose** the overall behaviour. Concrete classes, implementing the interface, define the variants' behaviour.



By 1995, it was clear that **[implementation] inheritance** was very easy to **overuse** and that **overuse** of inheritance was **very costly**.

[The Gang of Four] went so far as to stress:

“Favour object composition over class inheritance”



The Composite Reuse Principle

Prevents us from making **one of the most catastrophic mistakes** that **contribute to the demise of an object-oriented system**: **using inheritance as the primary reuse mechanism**

So we cut back on our use of **[implementation] inheritance**, often replacing it with **composition** or **delegation**.

[Template Method and Strategy are] two patterns that epitomize the difference between **inheritance** and **delegation**.

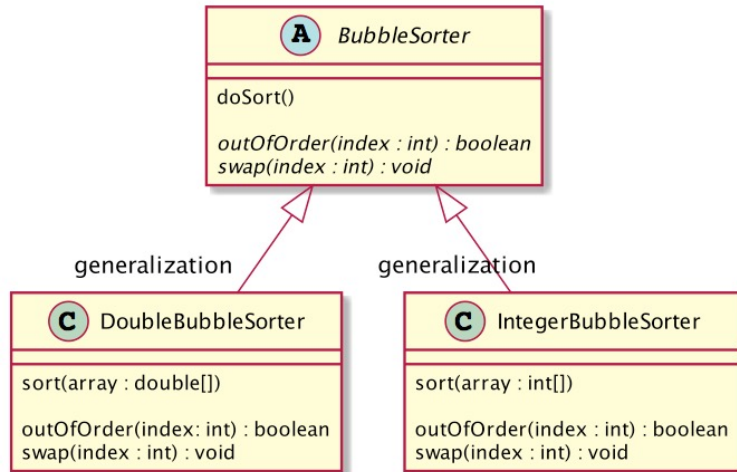
They solve similar problems and **can often be used interchangeably**



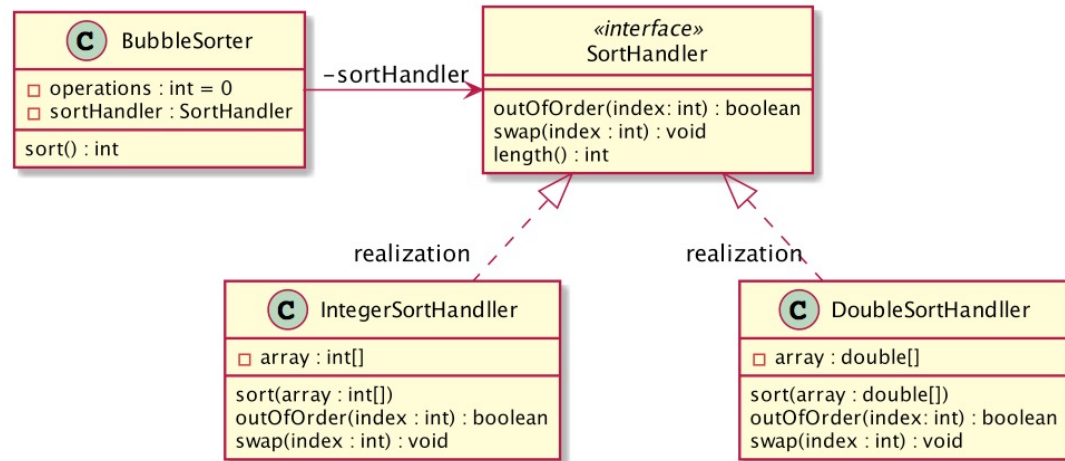


“These two patterns represent a **clear separation** of **generic** functionality from the **detailed implementation** of that functionality”.

Template Method Pattern



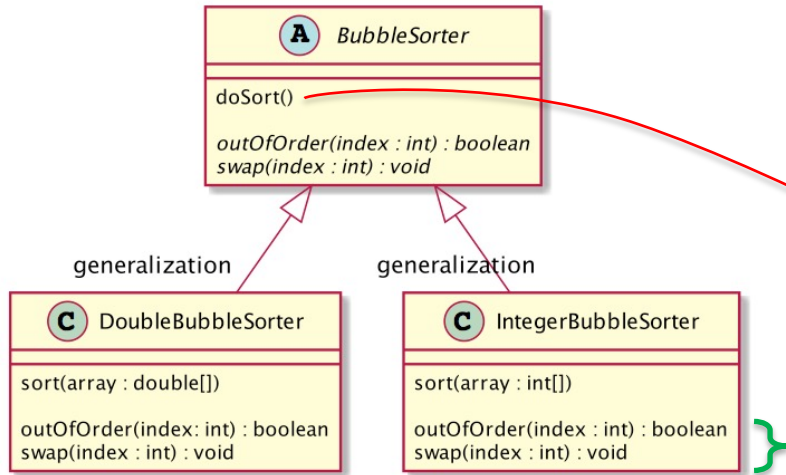
Strategy Pattern



In order to conform to the **DIP**, we want to make sure that the **generic algorithm** **does not depend on** the **detailed implementation**.

The **STRATEGY** pattern provides **one extra benefit** over the **TEMPLATE METHOD** pattern.

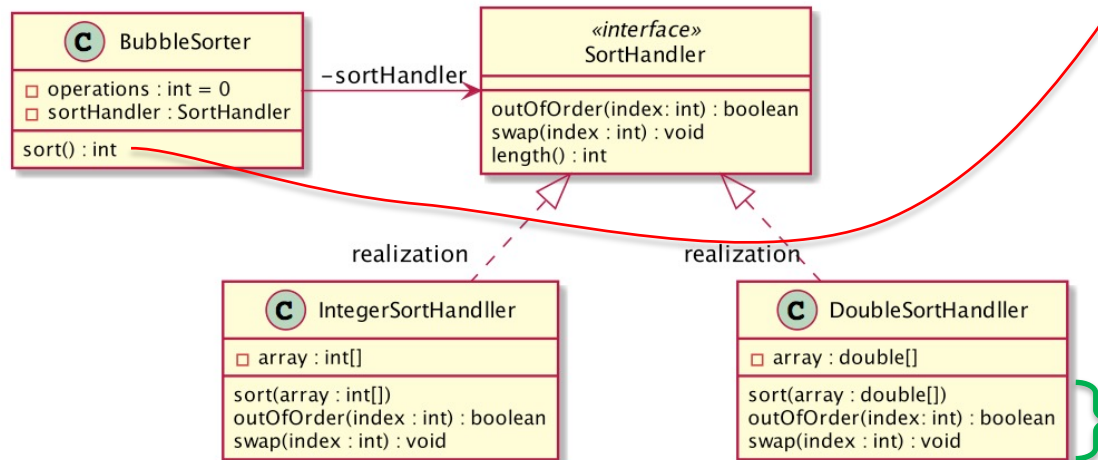
Template Method Pattern



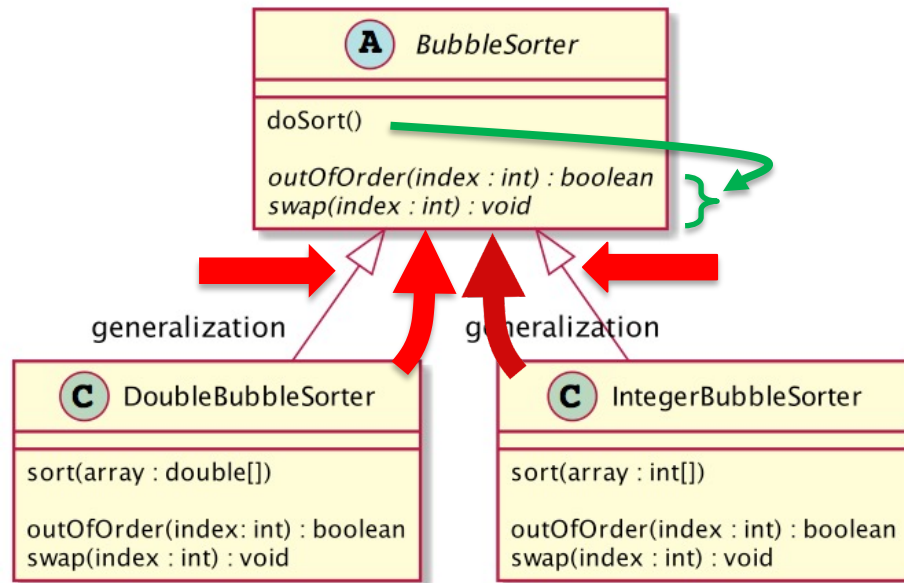
The sorting algorithm consists of:

- 1) Generic high level steps
Can be used to sort items of any type
- 2) Detailed operations/steps
Operate on items of a specific type

Strategy Pattern



Template Method Pattern



DIP



High-level modules **should not depend on** low-level modules.



Both should depend on **abstractions**.

Design Smell: **Immobility**



The **TEMPLATE METHOD** pattern allows a generic algorithm to manipulate many possible detailed implementations,

But **Template Method** **partially violates** the **DIP** because it uses **implementation inheritance**

so the detailed implementations **don't depend on an abstraction**

they depend on the generic algorithm

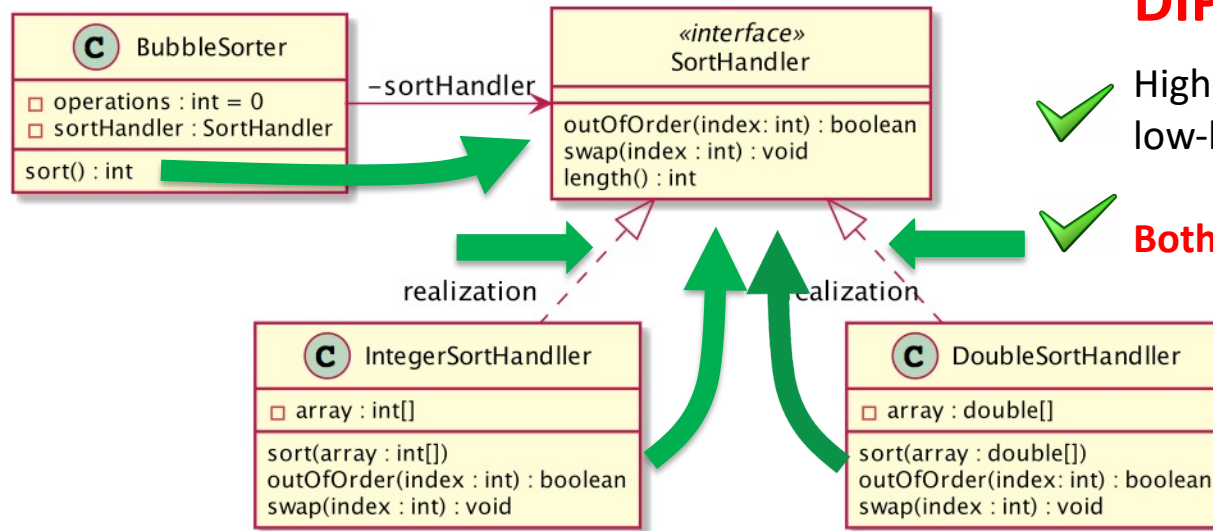
and so they are **inextricably bound** to it

and **cannot be reused by other generic algorithms**

BubbleSorter's **doSort()** method satisfies the **DIP** because it depends on abstract methods **outOfOrder()** and **swap()**

DoubleBubbleSorter and **IntegerBubbleSorter** do not satisfy the **DIP** because they depend on **BubbleSorter**, which is **NOT an abstraction** since it contains a concrete generic algorithm

Strategy Pattern



DIP

✓ High-level modules **should not depend on** low-level modules.

✓ **Both** should depend on **abstractions**.

The **Strategy** pattern **fully conforms** to the **DIP** because it uses **interface inheritance**

so the **detailed implementations** do **depend** on an **abstraction** (the **interface**),

so the **detailed implementations** can be manipulated by (reused for) **many different generic** algorithms

Not only does **BubbleSorter** satisfy the **DIP**, because its **sort()** method depends on interface **SortHandler**, i.e. an abstraction

but **IntegerSortHandler** and **DoubleSortHandler** also satisfy the **DIP**, because they also depend on the **SortHandler** abstraction

Strategy has this additional benefit over Template Method

Original OCP



inherits from
(OO inheritance)

The original version of the
OCP used **implementation
inheritance**

Contemporary OCP



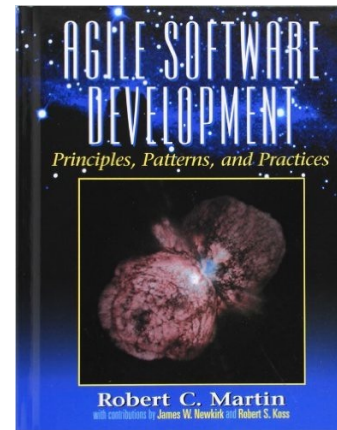
realization

+



generalization

While the contemporary version of the
OCP mostly uses **interface inheritance**,
it sometimes does use **implementation
inheritance**



In many ways, the OCP is at the heart of object-oriented design

Conformance to this principle is what yields the **greatest benefits** claimed for OO technology: **flexibility**, **reusability**, and **maintainability**

[it is not] a good idea to apply **rampant abstraction** to **every part** of the application.

Rather, it requires a dedication on the part of the developers to **apply abstraction only to those parts of the program that exhibit frequent change**.

Resisting premature abstraction is as important as abstraction itself.



I hope you enjoyed that.