Object Oriented Programming

Abstracts and interfaces

Methods - review

- Collection of **logically related** instructions
- Methods are used to perform certain **actions**
- Methods are referenced by **name** – message identifier
- Not independent components, **parts of objects**

Method declaration

```
<visiblity>
<type parameter>
<return type>
<method name>([formal parameters])
{
    <method body>
}
```

Class hierarchy - abstraction

`cat.say() => "Meow"`

`dog.say() => "Bark"`

`Abstraction`

- **`Mammal`** would be the parent class
- containing generic behavior **`say`**
- but what does this **Mammal.say** do?

On abstraction level of **`Mammal`**, **`say`** is not interpretable

Abstract behavior

When a behavior

- is expected on a **branch** of the class hierarchy – generic level
- is **introduced** on a certain level of abstraction – parent
- is **not interpretable** on that certain level – abstract

the behavior is said to be ***abstract***

Abstract method

By declaration, its existence is expected, but the implementation is not known.

The method has no exact implementation → has no method body

Has to be marked by **`abstract`** modifier

Method declaration

```
abstract
<visiblity>
<type parameter>
<return type>
<method name>([formal parameters]);
    // No body
```

Abstract class

A class which contains abstract methods, its **full implementation is not known.**

Abstract class

- Can **not** be **instantiated** – the implementation of a class is not defined
- To make such a class **work**, abstract behaviors have to be specified – **overridden**

  To give space to polymorphism, abstract class **has to be inherited**
- Has to be marked with **`abstract`** modifier

Abstract class

- As a class, a part of the class hierarchy – inheritance
- Can be target type of cast
- Can contain **abstract method** – but not necessary
- Can contain implemented (**non-abstract**) method – but not necessary

Class hierarchy - abstraction

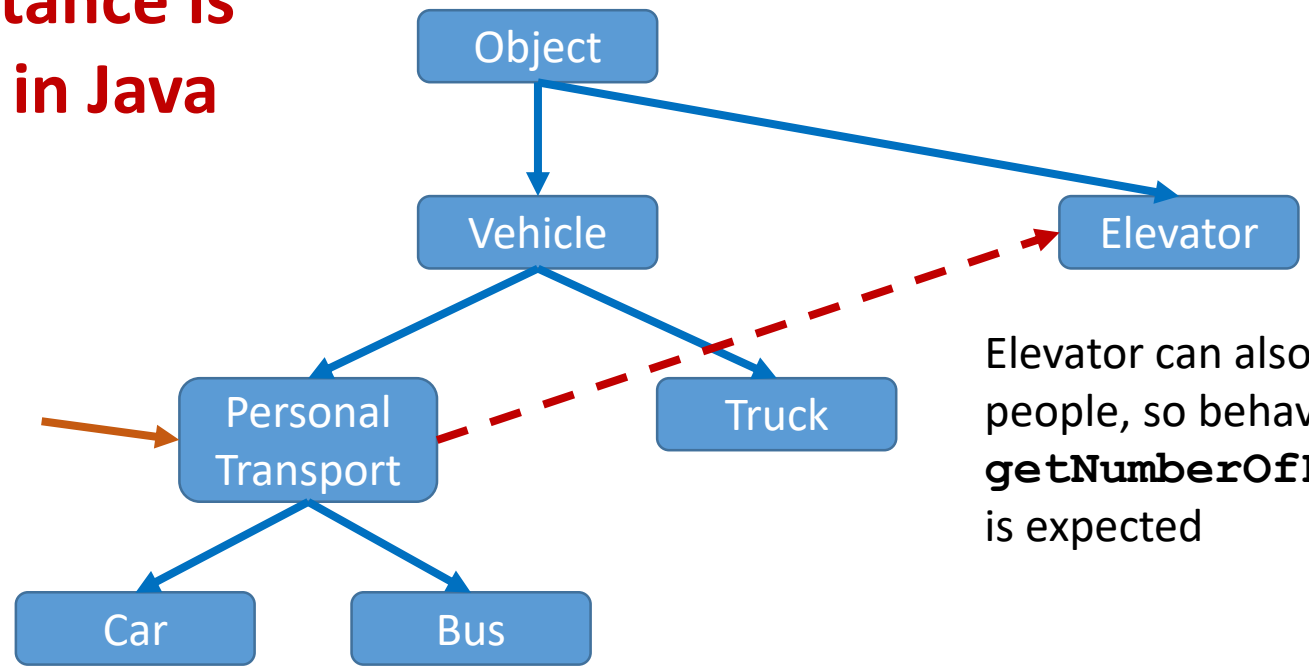
```
abstract class Mammal {
        abstract public String say();
}
                                class Cat extends Mammal {
                                    public String say() { return ”Meow”; }
                                }

            class Dog extends Mammal {
                public String say() { return ”Bark”; }
            }
```

Cast to abstract type

```
Cat cat = new Cat();
Dog dog = new Dog();

cat.say(); => ”Meow”
dog.say(); => ”Bark”

((Mammal)cat).say(); => ”Meow”          //using interface of
Mammal class
((Mammal)Dog).say(); => ”Bark”          //using interface of
Mammal class
```

Class hierarchy – multiple inheritance

Multiple inheritance is not supported in Java



Elevator should inherit PersonalTransport

behavior **`getNumberOfPersons`** is added to **`Vehicle`**

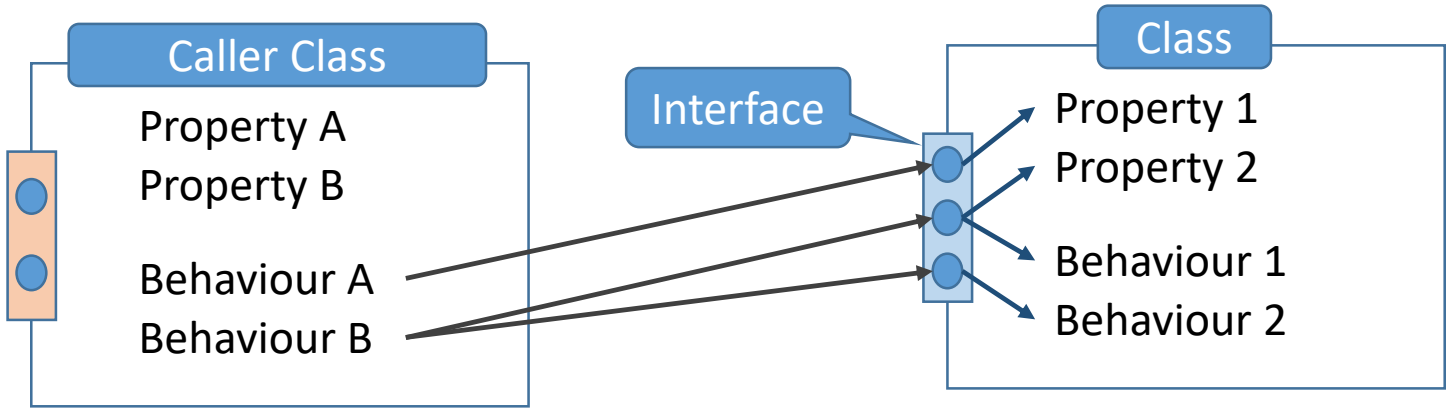
Elevator can also transport people, so behavior **`getNumberOfPersons`** is expected

Interface – review

Abstract, structural definition of a one direction connection surface.

Declares the usage modes of the program component – object .

A contract about expected behaviors of a class.



Interface definition of a **class**

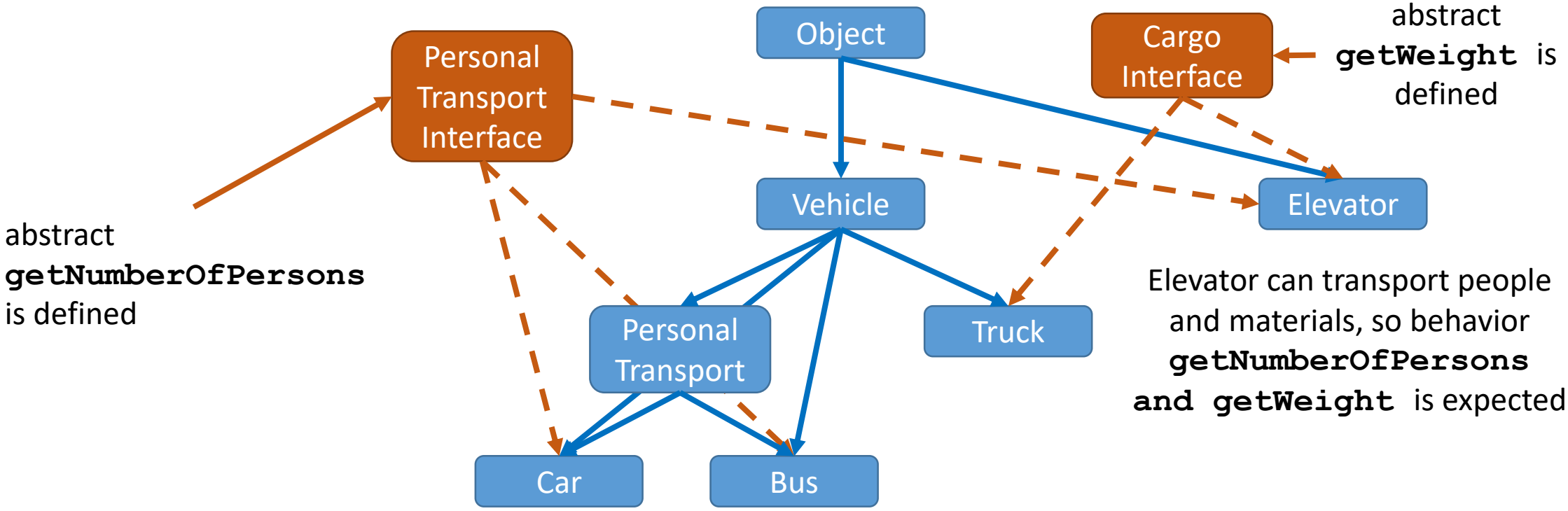
- By class developer (designer)
- Usage of class instances is specified
- By setting visibility modifiers

- Class interface contains all public components of the class (own and inherited)

Java interface

- A collection of abstract method definitions
- Can NOT be used to create objects – no instantiation
- Can be defined separately
- Not part of class hierarchy
- Interface can be implemented by any classes – `implements` keyword
- A class can **implement more interfaces**
- Can be used as a type for cast

Class hierarchy – with interface



Interface definition

```
interface PersonalTransport {
	public int getNumberOfPersons();
}
class Car extends Vehicle implements PersonalTransport {
	private int numberOfPersons;
	public int getNumberOfPersons() { return numberOfPersons; }
}
```

Abstract class vs Interface – I

- **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods.
- **Type of variables:** Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
- **Implementation:** Abstract class can provide the implementation of interface. Interface can’t provide the implementation of abstract class.

Abstract class vs Interface – II

- **Inheritance vs Implementation:** A Java interface can be implemented using keyword "`implements`" and abstract class can be extended using keyword "`extends`".
- **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- **Accessibility:** Members of a Java interface are public. An abstract class can have class members like private, protected, etc.

Cast a reference – review

- A reference of new type is created to the original object
- Cast changes the type of reference, not the original object
- Type of created reference specifies the access interface of the original object
- Casting in class hierarchy tree
  - on the branch of object
  - from type of object / the object which implements the interface
  - until the root

Cast reference

```
Car car = new Car();
Bus bus = new Bus();
PersonalTransport transporter;
transporter = (PersonalTransport)car;
transporter.getNumberOfPersons();
transporter = (PersonalTransport)bus;
transporter.getNumberOfPersons();
```

Class hierarchy design

- **Single Responsibility Principal**

  Members of a class are encapsulated by the responsibility zone.

- **Cohesion** – atomicity

  Methods of a class are stick together by common data and internal references.

- **Coupling** – dependency of classes

  Which part of the class hierarchy is affected by a change of a class

Cohesion and Coupling

Cohesion

A logical connection between data members and methods inside a class.

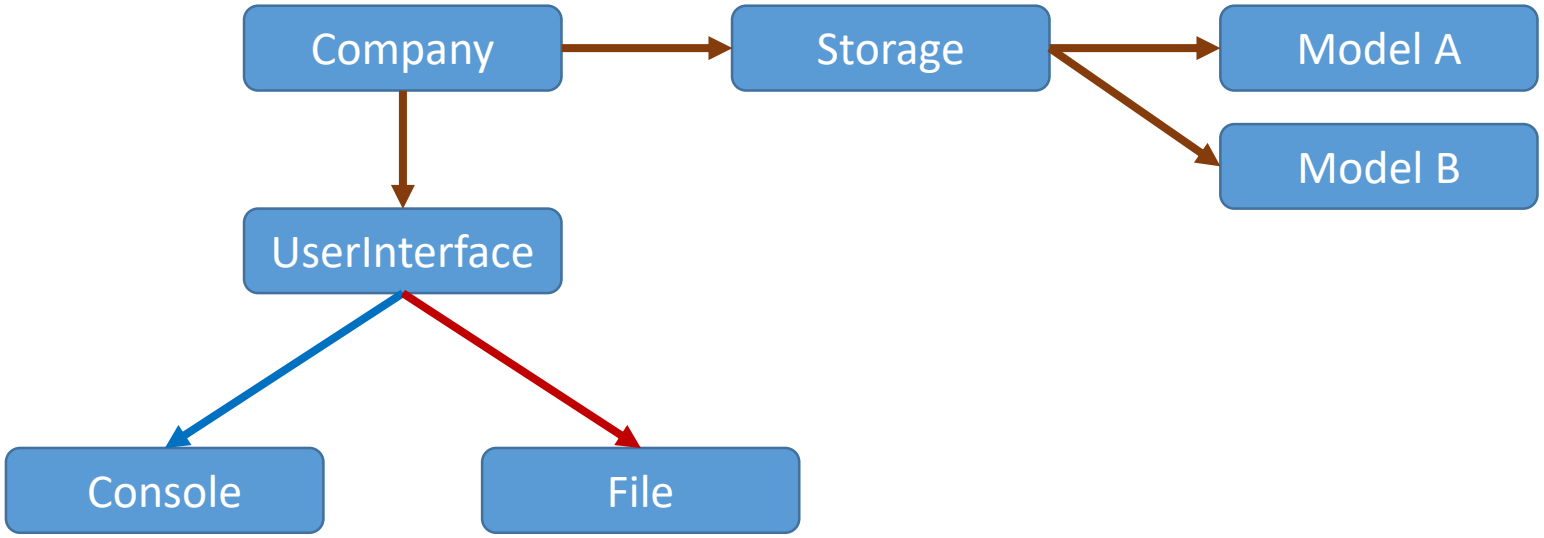
The base of encapsulation.

Coupling

Inter class dependency, how classes use resources of each other.

Client-server connections used for solving tasks are described, not the abstraction hierarchy.

Class dependency



Cohesion

A class should have only one task or responsibility (depending on abstraction level).

Base of cohesion and inherence is the responsibility!

When designing encapsulation and class hierarchy, consider the responsibility zones.

Not just with data and internal references, but also in connection with the operation.

On which functional change should the class also be changed?

Dependency – Coupling

Non-inherent, but cooperative classes are dependent and called **coupled**, when the type of server is known by the client.

The classes are independent, when the client knows only the interface required to complete a task (and that the server does implement is).

Cohesion and coupling

Cohesion

Represents the degree to which a part of a code base forms a logically single, atomic unit.

Degree of atomicity.

Coupling

represents the degree to which a single unit is independent from others.

It is the number of connections between units.

The fewer the number, the lower the coupling.

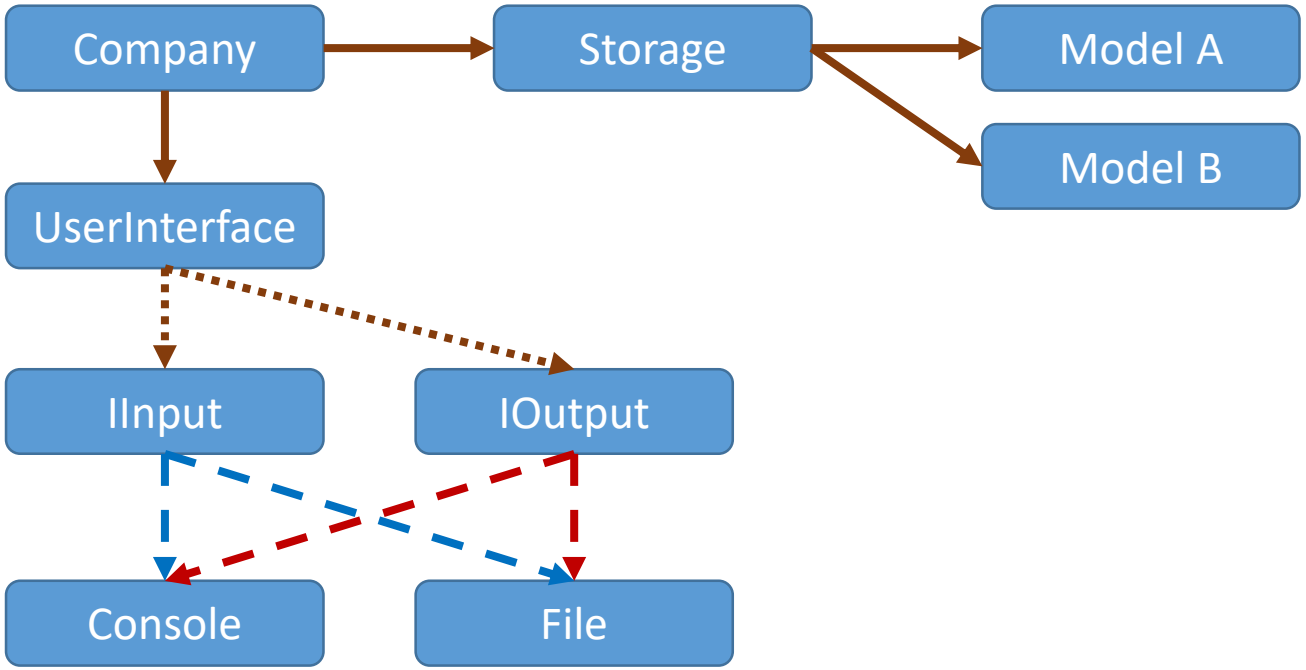
Decoupling

Its aim is to decrease the effect of classes on the dependency structure (graph)

Tools

- Use interface to describe connection – instead of class
- Inversion of control
- Dependency injection

Class dependency



Destructive decoupling

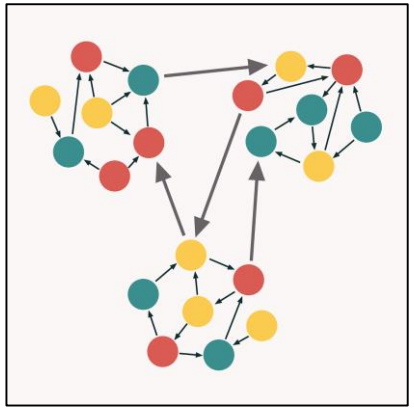
```
public class Order {
    public Order(IOrderLineFactory factory,

IOrderPriceCalculator calculator) {
        _factory = factory;
        _calculator = calculator;
    }
    public decimal Amount {
        get { return _calculator.CalculateAmount(_lines); }
    }
    public void AddLine(IProduct product, decimal price) {
        _lines.Add(_factory.CreateOrderLine(product, price));
    }
```

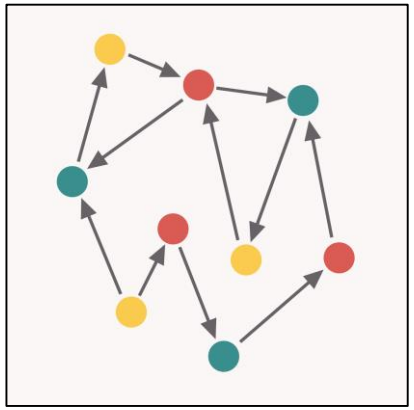
Ideal decoupling

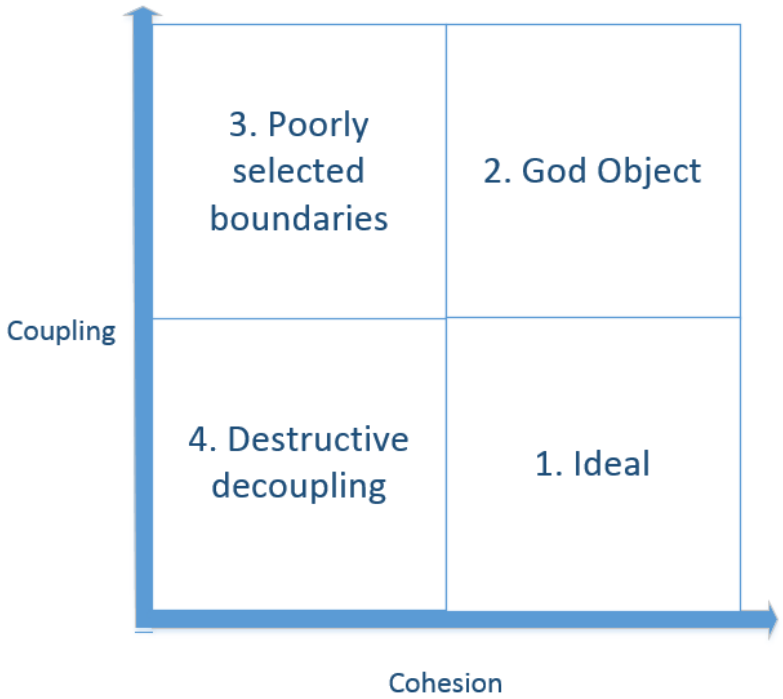
```
public class Order {
    public decimal Amount {
        get { return _lines.Sum(x => x.Price); }
    }

    public void AddLine(Product product, decimal amount) {
        _lines.Add(new OrderLine(product, amount));
    }
}
```

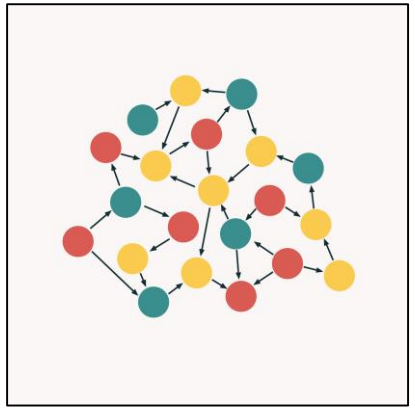
Decoupling harms readability/sustainability

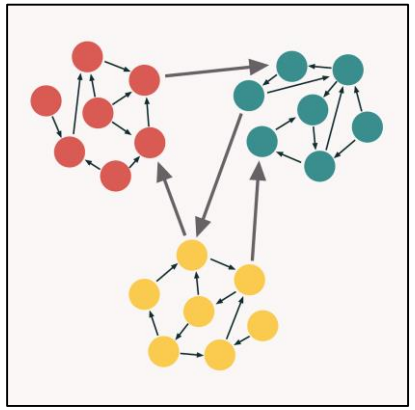
Cohesion and Coupling



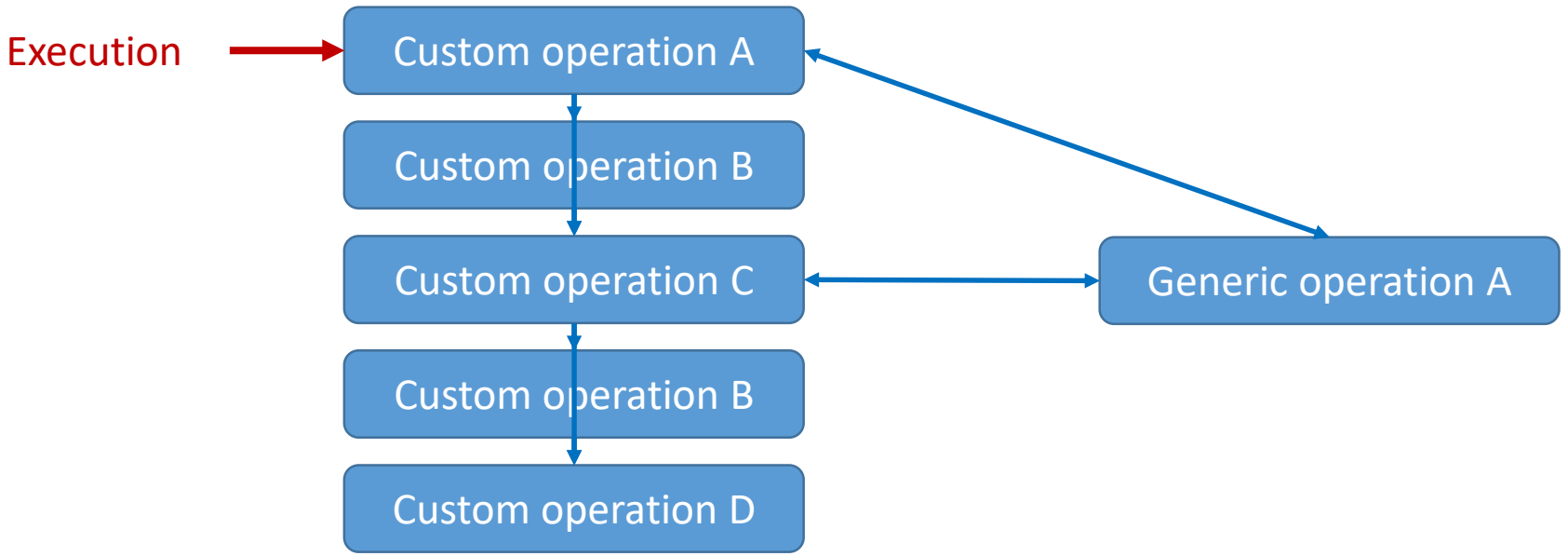








Generic program flow



Inversion of control

Custom-written portions of a computer program receive the flow of control from a generic framework.