



# Lecture 07: Factory Pattern

## IN710: Object-Oriented Systems Development

### Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Tuesday, 10 March

# LECTURE 06: OBSERVER PATTERN RECAP

- ▶ Design pattern 02: observer pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ Real world analogy
  - ▶ UML & implementation
  - ▶ Pros & cons

# LECTURE 07: FACTORY PATTERN TOPICS

- ▶ Design pattern 03: factory pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ UML & implementation
  - ▶ Applicability
  - ▶ Pros & cons

# FACTORY PATTERN: GoF

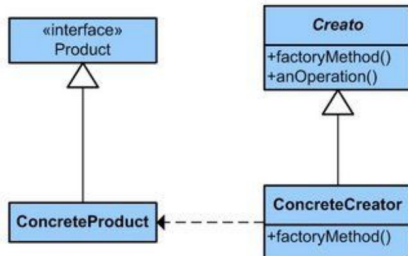
## ► GoF definition & UML

### Factory Method

**Type:** Creational

**What it is:**

Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



# FACTORY PATTERN: DEFINITION

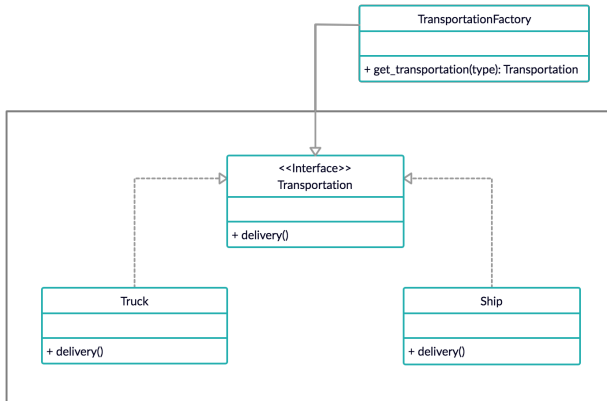
- ▶ Creational pattern
- ▶ Virtual constructor
- ▶ Deals with the problem of creating objects without having to specify the exact class of the object that will be created
- ▶ Done by creating objects by calling a factory method
  - ▶ Specified in an interface & implemented by child classes
  - ▶ Implemented in a base class & optionally overridden by derived classes
- ▶ Relies on inheritance

# FACTORY PATTERN: PROBLEM

- ▶ Logistics management application

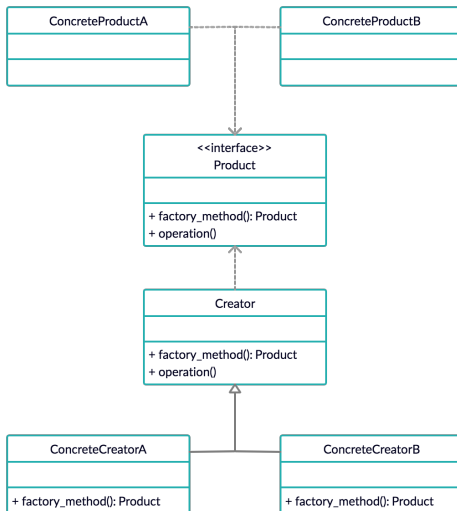
# FACTORY PATTERN: SOLUTION

- Factory class
- Interface class



# FACTORY PATTERN: UML

- Consider the following UML diagram:





# FACTORY PATTERN: IMPLEMENTATION

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print('Drawing_a_circle')

class Triangle(Shape):
    def draw(self):
        print('Drawing_a_triangle')

class ShapeFactory:
    def get_shape(self, type):
        return Circle() if type == 'circle' else Triangle()

def main():
    shape_factory = ShapeFactory()
    circle = shape_factory.get_shape('circle')
    triangle = shape_factory.get_shape('triangle')
    circle.draw()
    triangle.draw()

if __name__ == '__main__':
    main()
```

# FACTORY PATTERN: IMPLEMENTATION

```
from abc import ABC, abstractmethod

class ShapeFactory(ABC):
    @abstractmethod
    def factory_method(self):
        pass

    def draw(self):
        return self.factory_method().draw()

class CircleFactory(ShapeFactory):
    def factory_method(self):
        return Circle()

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print('Drawing a circle')

def main():
    circle_factory = CircleFactory()
    circle_factory.draw()

if __name__ == '__main__':
    main()
```

# FACTORY PATTERN: APPLICABILITY

- ▶ IDbCommand.CreateParameter - ADO.NET
- ▶ createElement - HTML5 DOM API
- ▶ javax.xml.parsers - Java
- ▶ QMainWindow::createPopupMenu - Qt

# FACTORY PATTERN: PROS

- ▶ Avoid coupling between the creator & concrete classes
- ▶ New products can be introduced without having to change the client's code
- ▶ All the creation code can be in one place in the program

## FACTORY PATTERN: CONS

- ▶ Application is complicated as new subclasses are introduced

# PRACTICAL

- ▶ Series of tasks covering today's lecture
- ▶ Worth 1% of your final mark for the Object-Oriented Systems Development course
- ▶ Deadline: Tuesday, 24 March at 5pm

# EXAM 01 RESULTS

- ▶ Check **Grades** tab in Microsoft Teams
- ▶ Average mark: 45.37

# LECTURE 08: SINGLETON PATTERN TOPICS

- ▶ Design pattern 04: singleton pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ Real world analogy
  - ▶ UML & implementation
  - ▶ Pros & cons