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
  - Strong vs. weak reference
  - Pros & cons

Lecture 07: Factory Pattern Topics

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

Factory Pattern: Definition

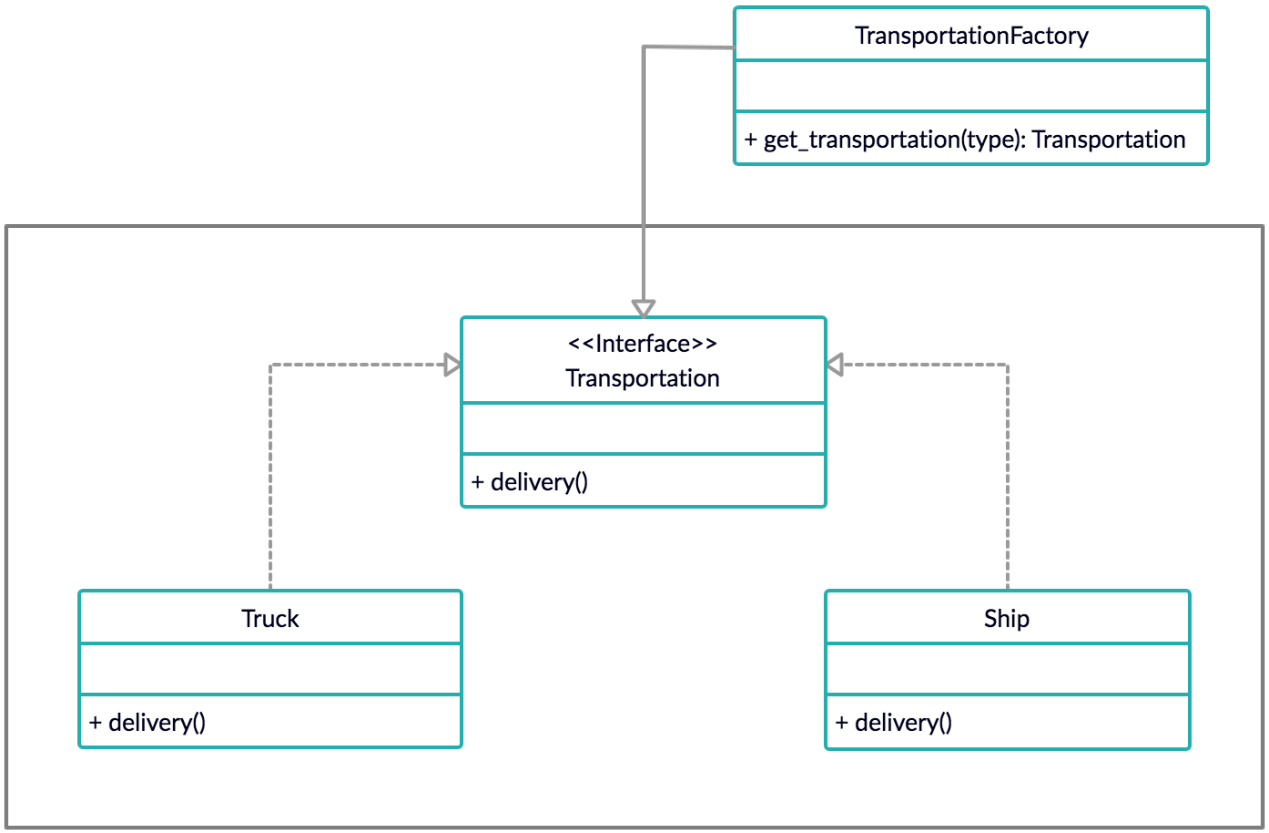
- Creational pattern
- Virtual constructor
- Deals with the problem of creating objects without having to specify the exact class of the object that will be created
- Done by creating objects by calling a factory method
  - Specified in an interface & implemented by child classes
  - Implemented in a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

Factory Pattern: Problem

- Logistics management application

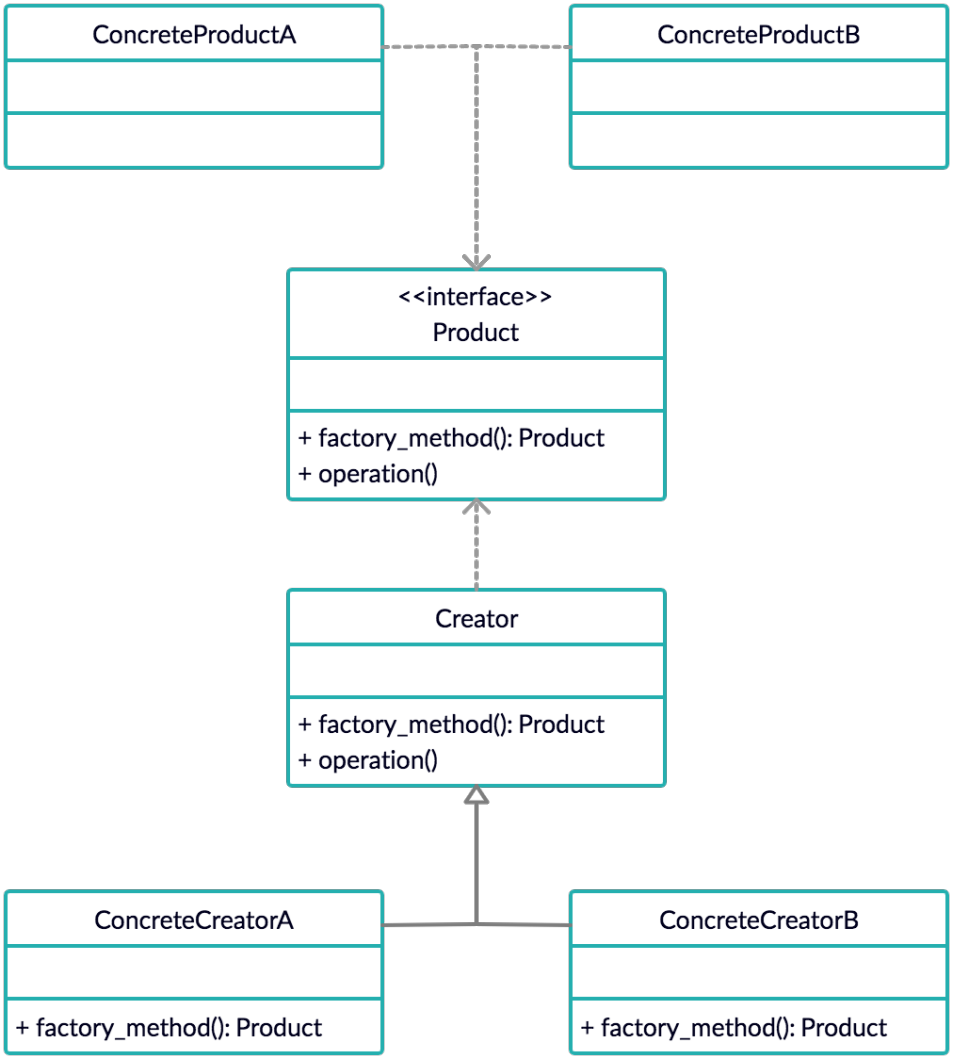
Factory Pattern: Solution

- 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

-

Lecture 08: Singleton Pattern Topics

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