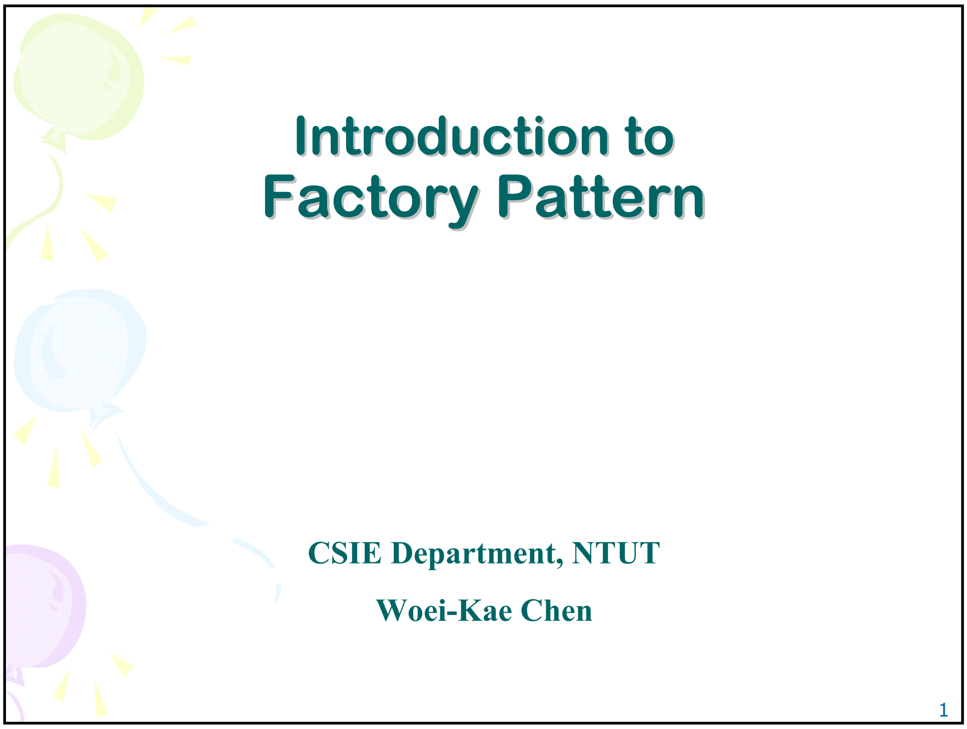
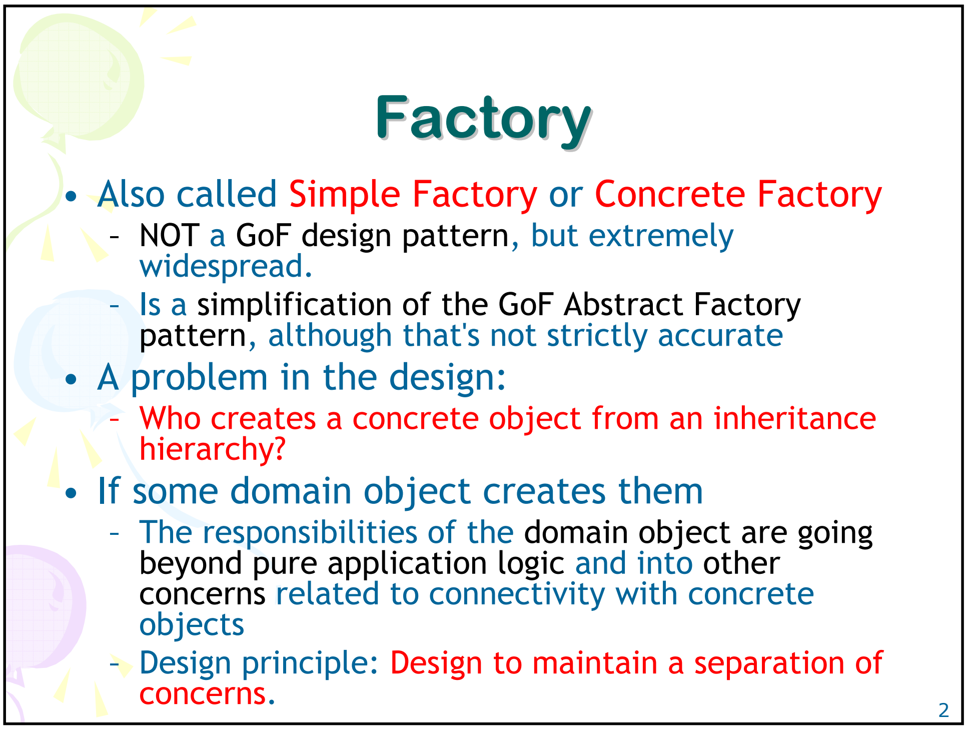


Introduction to Factory Pattern

CSIE Department, NTUT

Woei-Kae Chen

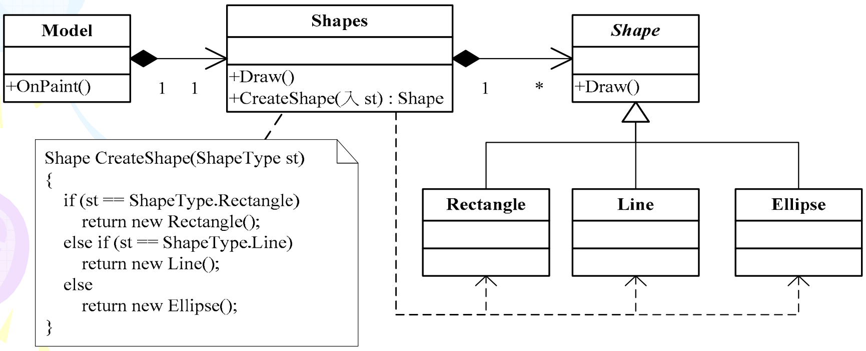


Factory

- Also called Simple Factory or Concrete Factory
  - NOT a GoF design pattern, but extremely widespread.
  - Is a simplification of the GoF Abstract Factory pattern, although that's not strictly accurate
- A problem in the design:
  - Who creates a concrete object from an inheritance hierarchy?
- If some domain object creates them
  - The responsibilities of the domain object are going beyond pure application logic and into other concerns related to connectivity with concrete objects
  - Design principle: Design to maintain a separation of concerns.

Factory

- Example
  - For a drawing tool, by information expert and low coupling, it is “shapes” that should be responsible for creating concrete shapes (e.g., Line)
  - A possible design is to have a CreateShape Method in Shapes.



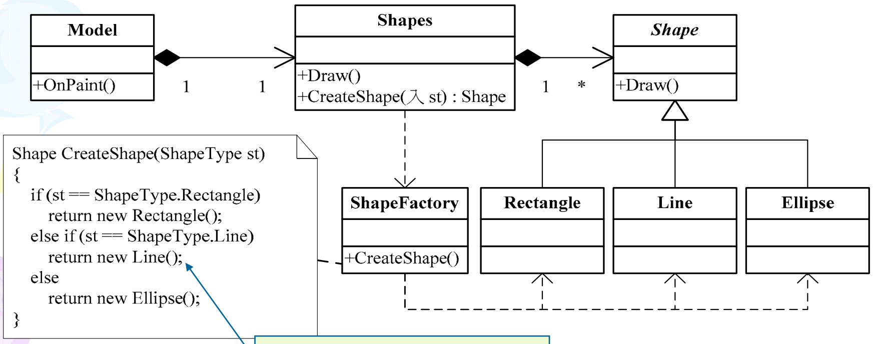
```
Shape CreateShape(ShapeType st)
{
   if (st == ShapeType.Rectangle)
       return new Rectangle();
   else if (st == ShapeType.Line)
       return new Line();
   else
       return new Ellipse();
}
```

Factory

- Design to maintain a separation of concerns
  - Modularize or separate distinct concerns into different areas, so that each has a cohesive purpose
  - Is an application of the GRASP High Cohesion principle
- If a domain object (such as a Shapes) to create the concrete shape
  - Does not support the goal of a separation of concerns
  - Increase coupling between Shapes and concrete Shape
  - Lowers its cohesion
- A common alternative in this case is to apply the **Factory** pattern
  - A Pure Fabrication "factory" object is defined to create objects.
- Factory objects have several advantages:
  - Separate the responsibility of complex creation into cohesive helper objects.
  - Hide potentially complex creation logic.
  - Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

Factory

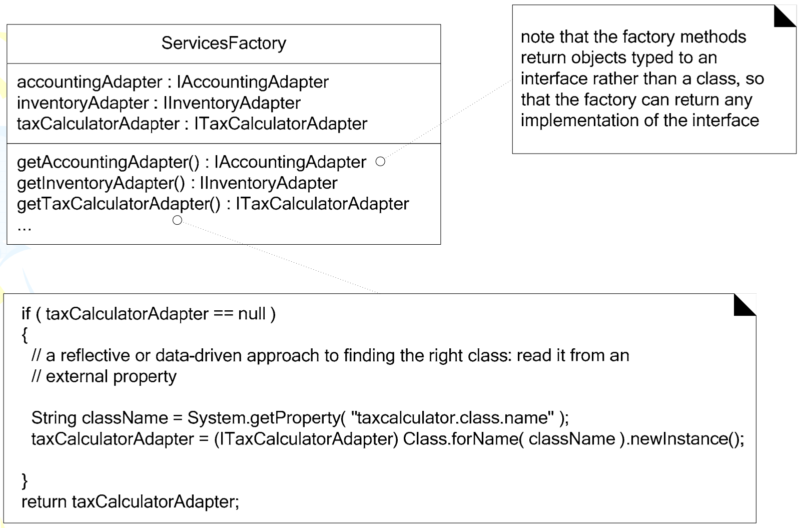
- Solution: Use a Factory object
  - To hide and localize complicated creation logic



```
Shape CreateShape(ShapeType st)
{
    if (st == ShapeType.Rectangle)
        return new Rectangle();
    else if (st == ShapeType.Line)
        return new Line();
    else
        return new Ellipse();
}
```

Use reflection to avoid if/switch

Factory


ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...

note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

```
if ( taxCalculatorAdapter == null )
{
  // a reflective or data-driven approach to finding the right class: read it from an
  // external property

  String className = System.getProperty( "taxcalculator.class.name" );
  taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();

}
return taxCalculatorAdapter;
```

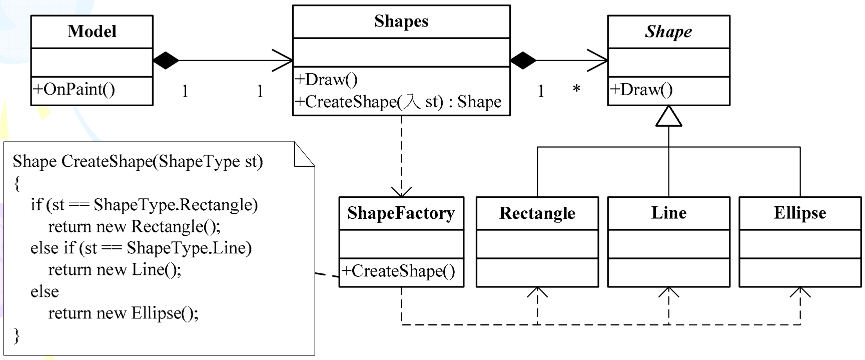
The Factory pattern

Factory

- Name: Factory
- Problem:
  - Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth?
- Solution: (advice)
  - Create a Pure Fabrication object called a Factory that handles the creation
- Note
  - In the ServicesFactory, the logic to decide which class to create is resolved by reading in the class name from an external source (e.g., via a system property) and then dynamically loading the class
  - This is an example of a partial data-driven design.
  - This design achieves Protected Variations with respect to changes in the implementation class of the adapter
- Related Patterns
  - Factories are often accessed with the Singleton pattern

Factory

- Implementation (C++)
  - Who is responsible for the delete operation?
  - Virtual Destructor?



```
Shape CreateShape(ShapeType st)
{
    if (st == ShapeType.Rectangle)
        return new Rectangle();
    else if (st == ShapeType.Line)
        return new Line();
    else
        return new Ellipse();
}
```