Comp-304 : Factory
Lecture 31

Alexandre Denault
Original notes by Hans Vangheluwe
Computer Science
McGill University
Fall 2007
10 / 23 = 43.5%
The following classes are from a real time strategy game where Humans and Orcs face each other for supremacy.

- Each Human unit has an Orcs counterpart which is identical.
The interface for players playing either race is identical.

Thus, every function that creates a unit has a similar piece of code:

```java
Worker worker;
if (player.race == RACE.HUMAN) {
    worker = createPeasants();
} else {
    worker = createPeon();
}
```

This is bad because

- It's code duplication.
- It's going to make things complicated when I add another race.

What can I do to avoid this?
Factory patterns are examples of creational patterns. They hide how objects are created and help make the overall system independent of how its objects are created and composed.
Two Types

- Class creational patterns focus on the use of inheritance to decide the object to be instantiated
  - Factory Method
- Object creational patterns focus on the delegation of the instantiation to another object
  - Abstract Factory
Abstract Factory

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Use the Abstract Factory pattern in any of the following situations:

- A system should be independent of how its products are created, composed, and represented
- A class can't anticipate the class of objects it must create
- A system must use just one of a set of families of products
- A family of related product objects is designed to be used together, and you need to enforce this constraint
Families of Soldiers

```
«interface»
AbstractFactory
    createWorker(): Worker
    createSoldier(): Soldier
    createCavalry(): Cavalry

HumanFactory
    createWorker(): Worker { return new HumanPeasant() }
    createSoldier(): Soldier { return new HumanFootman() }
    createCavalry(): Cavalry { return new HumanKnight() }

OrcFactory
    createWorker(): Worker { return new OrcPeon() }
    createSoldier(): Soldier { return new OrcGrunt() }
    createCavalry(): Cavalry { return new OrcWolfRider() }
```
Participants

- **AbstractFactory**
  - Declares an interface for operations that create abstract product objects

- **ConcreteFactory**
  - Implements the operations to create concrete product objects

- **AbstractProduct**
  - Declares an interface for a type of product object

- **ConcreteProduct**
  - Defines a product object to be created by the corresponding concrete factory
  - Implements the AbstractProduct interface

- **Client**
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes
Consequences

■ Exchanging or adding product families is easy.
■ It also promotes consistencies among product (across families).
■ However, adding new products involves a lot more modifications.
■ Before 3D acceleration, GUI system in game very sensitive to screen resolution variations.
■ For gameplay reasons, whatever the screen resolution, the GUI had to be the same size.
■ Because of this complexity, many games had only one resolution.
GUIFactory

AbstractGuiFactory
createWindow(): Window
createButton(): Widget
createLabel(): Widget
createTextBox(): Widget
createFrame(): Widget

Gui640x480Factory
createWindow(): Window
createButton(): Widget
createLabel(): Widget
createTextBox(): Widget
createFrame(): Widget

Gui800x600Factory
createWindow(): Window
createButton(): Widget
createLabel(): Widget
createTextBox(): Widget
createFrame(): Widget

Gui1024x768Factory
createWindow(): Window
createButton(): Widget
createLabel(): Widget
createTextBox(): Widget
createFrame(): Widget
Factories as Singletons

- Typically, you only need one instance of a factory per product family.
- That makes it an ideal candidate for Singleton.
Extensible Factories

■ One of the big limitation of Abstract Factory is the impact of adding new products.
■ A flexible, but less safe design, is to parameterize the object you want to create.
As already mentioned, this is not a safe design.
  • Implementing in all factories
  • Coercision

In addition, all return Products must have the same return type.
Another Example

```
<<interface>>
DocumentGenerator
createLetter(): Letter
createFax(): Fax
createResume(): Resume
createCoverPage(): CoverPage
```

```
BlackWhiteDocumentGenerator
createLetter(): Letter
createFax(): Fax
createResume(): Resume
createCoverPage(): CoverPage
```

```
ColorDocumentGenerator
createLetter(): Letter
createFax(): Fax
createResume(): Resume
createCoverPage(): CoverPage
```
I'm currently designing a unified driver for Nvidia Geforce cards.

This unified driver supports the following cards.

- Geforce 2
- Geforce 3
- Geforce 4
- Geforce FX
- Geforce 6
- Geforce 7
- Geforce 8
Shaders are programs written specifically for graphic cards to perform visual effects.

Two main types of shaders exist:
- Pixel shaders: works on a 2D image / texture
- Vertex shaders: works on a 3D mesh
Different architectures support different types of shaders.

- Geforce 2, 3, 4: Pixel and Vertex Shaders 1.0
- Geforce FX: Pixel and Vertex Shaders 2.0
- Geforce 6, 7: Pixel and Vertex Shaders 3.0
- Geforce 8: Pixel and Vertex Shaders 4.0
Shader Objects

- Interface Shader
  - Interface PixelShader
    - PixelShader1
    - PixelShader2
    - PixelShader3
    - PixelShader4
  - Interface VertexShader
    - VertexShader1
    - VertexShader2
    - VertexShader3
    - VertexShader4
Creating these objects

- As already mentioned, different cards create different types of shader objects.
  - If a particular functionality is not supported by a particular card, it is sometimes emulated in software.

- However, an OpenGL or DirectX application should be able to create shader objects in a generic fashion.
  - i.e. It doesn't need to know we have a GeForce FX.
 ShaderFactory

ShaderManager
getShaderFactory(): ShaderFactory

interface ShaderFactory
createPixelShader(): PixelShader
createVertexBuffer(): VertexShader

Shader1Factory
createPixelShader(): PixelShader { return new PixelShader1() }
createVertexBuffer(): VertexShader { return new VertexShader1() }

Shader2Factory
createPixelShader(): PixelShader { return new PixelShader2() }
createVertexBuffer(): VertexShader { return new VertexShader2() }

...