Comp-304 : Factory and Proxy Patterns

Alexandre Denault
Original notes by Hans Vangheluwe
Computer Science
McGill University
Winter 2009
Mercury class evaluations
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
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

ShaderFactory

createPixelShader(): PixelShader
createVertexShader(): VertexShader

Shader1Factory

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

Shader2Factory

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

...
Game X takes 30 seconds to load a level.
  - It needs to load 300 images.
On average, a level only uses 50 images.
Images are stored in an FileImage class
  - FileImage is a proprietary class file from a 3rd party library
How can I improve performance and minimize code changes?

Original idea from Wikipedia
Image and FileImage

```
<interface>
  Image
</interface>

+draw(): void

FileImage

+FileImage(String path)
+draw(): void
```
Focus on DelayedImage

```java
if (this.image == null) {
    this.image = new FileImage(this.path);
}
this.image.draw();
```
■ Provide a surrogate or placeholder for another object to control access to it.

■ Aka: Surrogate
What is a proxy?

A proxy is

- a person authorized to act for another person
- an agent or substitute
- the authority to act for another
Sometimes, you want to modify the behavior of an object (or control access to it), without modifying the object itself.
Use the Proxy pattern when

- You want to remotely access a local object.
- You want to create expensive objects on demand.
- You want to protect the original object.
- You want to make the original object smarter.
How does it work?
Proxies are even easier to use (and transparent) in languages where you can …
- Override the member access operators.
- Use Aspects.
Proxy doesn't need to know the concrete type of the subject.
What to do with proxies without subjects?
Arithmetic Example
Mammoth Example
Replication Systems

Node 1 → Node 3 → Node 2
Node 1 → Node 3 → Node 4
Node 1 → Node 3 → Node 5
Node 2 → Node 4
Node 2 → Node 5

= master object
= replica
Architecture

```
interface Player
+
setDestination(Point p): bool

PlayerImpl
+
setDestination(Point p): bool

MasterPlayerProxy
-subject : Player
+
setDestination(Point p): bool

DuplicaPlayerProxy
+
setDestination(Point p): bool

subject.setDestination(p)

network.send(
  target, setDestination, p)
bool b = network.waitFor()
return b
```
Local Call
Profiler Example

```
int tStart = time.now();
int retVal = subject.solve(formula);
int tEnd = time.now();
log(me, solve. tEnd - tStart);
return retVal;
```
Other Types of Proxies

- **Copy-On-Write Proxy** - Defers copying (cloning) a target object until required by client actions. Really a form of virtual proxy.
- **Protection (Access) Proxy** - Provides different clients with different levels of access to a target object.
- **Cache Proxy** - Provides temporary storage of the results of expensive target operations so that multiple clients can share the results.
- **Firewall Proxy** - Protects targets from bad clients (or vice versa).
- **Synchronization Proxy** - Provides multiple accesses to a target object.
- **Smart Reference Proxy** - Provides additional actions whenever a target object is referenced such as counting the number of references to the object.
Extending 3rd party tools

- You don't have the code.
- Can't always subclass existing code.
- Proxies are the next best things.
Design Patterns, the big picture

- Design Patterns are solutions to problems.
- You can teach them with lectures, but that's not optimal.
- To appreciate them, you need to use them.
- And first them in using them is recognizing when to use them.
Name that Design Pattern
Left vs Right Edition

<insert inspirational music here>
I need to efficiently update several displays monitoring the content of a file, every time the file is updated.
I need to be able to queue actions, since I can only process one at a time.
I need to generate objects from either the Fire, Water, Air or Earth family of objects.
I need to traverse a data structure containing Ninja, Monkey, Pirates and Zombie objects.
I need to make sure that a maximum of three copies of my SuperCache object exists.
I need to make a library I just purchased type compatible with an existing one.
I need to add security to an object from the library I just purchased.
I need my application to process all orders in an identical fashion, regardless if they have sub-orders or not.