Abstract Factory in C++: Before and after

Read full article

See the code

01 Trying to maintain portability across multiple “platforms” routinely requires lots of preprocessor “case” statements. The Factory pattern suggests defining a creation services interface in a Factory base class, and implementing each “platform” in a separate Factory derived class.

Before

02 The client creates “product” objects directly, and must embed all possible platform permutations in nasty looking code.

03

```
#define MOTIF
```

```
class Widget {
public:
   virtual void draw() = 0;
};

class MotifButton : public Widget {
public:
   void draw() { cout << "MotifButton\n"; }
};
class MotifMenu : public Widget {
public:
   void draw() { cout << "MotifMenu\n"; }
};

class WindowsButton : public Widget {
public:
   void draw() { cout << "WindowsButton\n"; }
};
class WindowsMenu : public Widget {
public:
   void draw() { cout << "WindowsMenu\n"; }
};

void display_window_one() {
#ifdef MOTIF
   Widget* w[] = { new MotifButton,
                   new MotifMenu };
#else // WINDOWS
   Widget* w[] = { new WindowsButton,
                   new WindowsMenu };
#endif
   w[0]->draw();  w[1]->draw();
}

void display_window_two() {
#ifdef MOTIF
```

```
   Widget* w[] = { new MotifMenu,
                   new MotifButton };
#else // WINDOWS
   Widget* w[] = { new WindowsMenu,
                   new WindowsButton };
#endif
   w[0]->draw();  w[1]->draw();
}

int main() {
#ifdef MOTIF
   Widget* w = new MotifButton;
#else // WINDOWS
   Widget* w = new WindowsButton;
#endif
   w->draw();
   display_window_one();
   display_window_two();
}
```

Output

```
MotifButton
MotifButton
MotifMenu
MotifMenu
MotifButton
```

After

04 The client: creates a platform- specific “factory” object, is careful to eschew use of “new”, and delegates all creation requests to the factory.

05

```
#define WINDOWS

class Widget {
public:
   virtual void draw() = 0;
};

class MotifButton : public Widget {
public:
   void draw() { cout << "MotifButton\n"; }
};
class MotifMenu : public Widget {
public:
   void draw() { cout << "MotifMenu\n"; }
};

class WindowsButton : public Widget {
public:
   void draw() { cout << "WindowsButton\n"; }
};
class WindowsMenu : public Widget {
public:
   void draw() { cout << "WindowsMenu\n"; }
};

class Factory {
public:
   virtual Widget* create_button() = 0;
   virtual Widget* create_menu() = 0;
};

class MotifFactory : public Factory {
public:
   Widget* create_button() {
      return new MotifButton; }
   Widget* create_menu()   {
      return new MotifMenu; }
};

class WindowsFactory : public Factory {
public:
   Widget* create_button() {
      return new WindowsButton; }
   Widget* create_menu()   {
      return new WindowsMenu; }
};

Factory* factory;

void display_window_one() {
   Widget* w[] = { factory->create_button(),
                   factory->create_menu() };
   w[0]->draw();  w[1]->draw();
}

void display_window_two() {
   Widget* w[] = { factory->create_menu(),
                   factory->create_button() };
   w[0]->draw();  w[1]->draw();
}

int main() {
#ifdef MOTIF
   factory = new MotifFactory;
#else // WINDOWS
   factory = new WindowsFactory;
#endif

   Widget* w = factory->create_button();
   w->draw();
```

```
    display_window_one();
    display_window_two();
}
```

```
Output
WindowsButton
WindowsButton
WindowsMenu
WindowsMenu
WindowsButton
```

List of Abstract Factory examples

C# examples

- Abstract Factory in C#

C++ examples

- Abstract Factory in C++: Before and after <=[You are here]
- Abstract Factory in C++

Delphi examples

- Abstract Factory in Delphi

Java examples

- Abstract Factory in Java
- Abstract Factory in Java

PHP examples

- Abstract Factory in PHP

- Abstract Factory in PHP

‹ Creational patterns

↑ Abstract Factory

Builder Design Pattern ›