**An Example Where It Is Needed:**Consider a user interface toolkit that supports multiple look-and-feel standards,

such as Motif and Presentation Manager. Different look-and-feels define different

appearances and behaviors for user interface "widgets" like scroll bars, windows,

and buttons. To be portable across look-and-feel standards, an application should

not hard-code its widgets for a particular look and feel. Instantiating

look-and-feel-specific classes of widgets throughout the application makes it

hard to change the look and feel later.

We can solve this problem by defining an abstract WidgetFactory class that declares

an interface for creating each basic kind of widget. There's also an abstract

class for each kind of widget, and concrete subclasses implement widgets for

specific look-and-feel standards. WidgetFactory's interface has an operation that

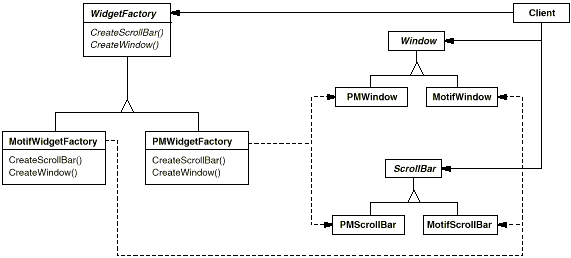
returns a new widget object for each abstract widget class. Clients call these

operations to obtain widget instances, but clients aren't aware of the concrete

classes they're using. Thus clients stay independent of the prevailing look and

feel.

(Operations are interface.clients call the operation. Now, those operations instantiate the classes accordingly. i.e. clients are not aware of the concrete subclass used)



There is a concrete subclass of WidgetFactory for each look-and-feel standard.

Each subclass implements the operations to create the appropriate widget for the

look and feel. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the

corresponding operation on the PMWidgetFactory returns a scroll bar for

Presentation Manager. Clients create widgets solely through the WidgetFactory

interface and have no knowledge of the classes that implement widgets for a

particular look and feel. In other words, clients only have to commit to an interface

defined by an abstract class, not a particular concrete class.

A WidgetFactory also enforces dependencies between the concrete widget classes.

A Motif scroll bar should be used with a Motif button and a Motif text editor,

and that constraint is enforced automatically as a consequence of using a

MotifWidgetFactory.

**When The Abstract Factory Pattern is used?**

Use the Abstract Factory pattern when

* a system should be independent of how its products are created, composed,

and represented.

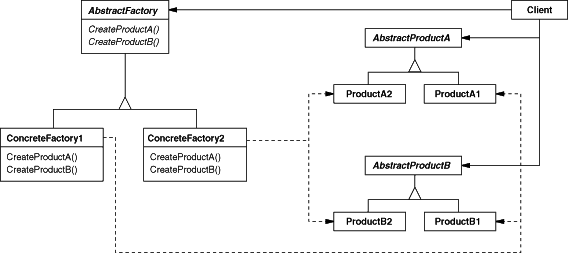
* a system should be configured with one of multiple families of products.
* a family of related product objects is designed to be used together, and

you need to enforce this constraint.

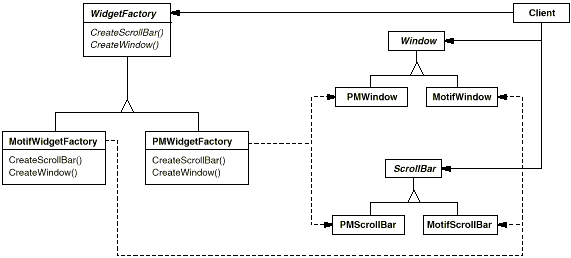
* you want to provide a class library of products, and you want to reveal

just their interfaces, not their implementations.

**Structure:**

**Generic AbstractFactory Version:**

**Specialized Version: The Example:**



Now, let’s discuss the participants:

* **AbstractFactory (WidgetFactory)**
  + declares an interface for operations that create abstract product objects.
* **ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)**
  + implements the operations to create concrete product objects. (It implements the operations provided by AbstractFactory)
* **AbstractProduct (Window, ScrollBar)**
  + declares an interface for a type of product object.
* **ConcreteProduct (MotifWindow, MotifScrollBar)**
  + 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.

**Implementation:**

1. Factories as singletons. An application typically needs only one instance

of a ConcreteFactory per product family. So it's usually best implemented

as a Singleton.

1. Creating the products. AbstractFactory only declares an interface for

creating products. It's up to ConcreteProduct subclasses to actually create

them. The most common way to do this is to define a factory method (see

Factory Method for each product. A concrete factory will specify Creating the products. AbstractFactory only declares an interface for creating products. It's up to ConcreteProduct subclasses to actually create them. The most common way to do this is to define a factory method for each product. A concrete factory will specify

1. If many product families are possible, the concrete factory can be implemented using the Prototype pattern. The concrete factory is initialized with a prototypical instance of each product in the family, and it creates a new product by cloning its prototype. The Prototype-based approach eliminates the need for a new concrete factory class for each new product family.

**Now, consider the following implementation (again, this is not the implementation which is right, actually this is a wrong implementation, we just need that to express what are the wrong things in this implementation and how should we correct those in actual implementation of AbstractFactory)**

#include <iostream>

#define LINUX

using namespace std;

/\*\*

\* Abstract base product.

\*/

class Widget

{

public:

virtual void draw() = 0;

//a pure virtual function making it abstract class

};

/\*\*

\* Concrete product family 1.

\*/

class LinuxButton : public Widget

{

public:

void draw() { cout << "LinuxButton\n"; }

};

class LinuxMenu : public Widget

{

public:

void draw() { cout << "LinuxMenu\n"; }

};

/\*\*

\* Concrete product family 2.

\*/

class WindowsButton : public Widget

{

public:

void draw() { cout << "WindowsButton\n"; }

};

class WindowsMenu : public Widget

{

public:

void draw() { cout << "WindowsMenu\n"; }

};

/\*\*

\* Here's a client, which uses concrete products directly.

\* It's code filled up with nasty switch statements

\* which check the product type before its use.

\*/

class Client

{

public:

void draw()

{

#ifdef LINUX

Widget \*w = new LinuxButton;

#else // WINDOWS

Widget \*w = new WindowsButton;

#endif

w->draw();

display\_window\_one();

display\_window\_two();

}

void display\_window\_one()

{

#ifdef LINUX

Widget \*w[] = {new LinuxButton,new LinuxMenu};

//list initialization

#else // WINDOWS

Widget \*w[] = {new WindowsButton,new WindowsMenu};

#endif

w[0]->draw();

w[1]->draw();

}

void display\_window\_two()

{

#ifdef LINUX

Widget \*w[] = {new LinuxButton,new LinuxMenu};

//list initialization

#else // WINDOWS

Widget \*w[] = {new WindowsButton,new WindowsMenu};

#endif

w[0]->draw();

w[1]->draw();

}

};

int main()

{

Client \*c = new Client();

c->draw();

return 0;

}

**Now, what’s the flaw of this approach?**

The client creates "product" objects directly, and must embed all possible platform permutations in nasty looking code. Here, it is just two products. Imagine there are situations where there are lots of products. Also, those switch statements are used a number of times. (in draw method, in display\_window\_one method, display\_window\_two method). Also, client should not have a direct control over creation of widget set for linux or windows .

Now, consider an implementation using the AbstractFactory Method:

#include <iostream>

#define LINUX

using namespace std;

/\*\*

\* Abstract base product. It should define an interface

\* which will be common to all products. Clients will

\* work with products through this interface, so it

\* should be sufficient to use all products.

\*/

class Widget

{

public:

virtual void draw() = 0;

};

/\*\*

\* Concrete product family 1.

\*/

class LinuxButton : public Widget

{

public:

void draw() { cout << "LinuxButton\n"; }

};

class LinuxMenu : public Widget

{

public:

void draw() { cout << "LinuxMenu\n"; }

};

/\*\*

\* Concrete product family 2.

\*/

class WindowsButton : public Widget

{

public:

void draw() { cout << "WindowsButton\n"; }

};

class WindowsMenu : public Widget

{

public:

void draw() { cout << "WindowsMenu\n"; }

};

/\*\*

\* Abstract factory defines methods to create all

\* related products.

\*/

/\*Now, we will no longer be creating WindowsButton, WindowsMenu directly in case of windowsWidget set\*/

class Factory

{

public:

virtual Widget \*create\_button() = 0;

virtual Widget \*create\_menu() = 0;

};

/\*\*

\* Each concrete factory corresponds to one product

\* family. It creates all possible products of

\* one kind.

\*/

class LinuxFactory : public Factory

{

public:

Widget \*create\_button()

{

return new LinuxButton;

}

Widget \*create\_menu()

{

return new LinuxMenu;

}

};

/\*\*

\* Concrete factory creates concrete products, but

\* returns them as abstract.

\*/

class WindowsFactory : public Factory

{

public:

Widget \*create\_button()

{

return new WindowsButton;

}

Widget \*create\_menu()

{

return new WindowsMenu;

}

};

/\*\*

\* Client receives a factory object from its creator.

\*

\* All clients work with factories through abstract

\* interface. They don't know concrete classes of

\* factories. Because of this, you can interchange

\* concrete factories without breaking clients.

\*

\* Clients don't know the concrete classes of created

\* products either, since abstract factory methods

\* returns abstract products.

\*/

class Client

{

private:

Factory \*factory;

public:

Client(Factory \*f)

{

factory = f;

}

void draw()

{

Widget \*w = factory->create\_button();

w->draw();

display\_window\_one();

display\_window\_two();

}

void display\_window\_one()

{

Widget \*w[] = {factory->create\_button(),factory->create\_menu()};

//factory is the base class pointer.

w[0]->draw();

w[1]->draw();

}

void display\_window\_two()

{

Widget \*w[] = {factory->create\_menu(),factory->create\_button()};

w[0]->draw();

w[1]->draw();

}

};

/\*\*

\* Now the nasty switch statement is needed only once to

\* pick and create a proper factory. Usually that's

\* happening somewhere in program initialization code.

\*/

int main()

{

Factory \*factory;

#ifdef LINUX

factory = new LinuxFactory;

#else // WINDOWS

factory = new WindowsFactory;

#endif

//note that, factory will be always instantiated.

//because, conditional compilation is used So, compiled code does not haave any flaws

Client \*c = new Client(factory);

c->draw();

}

**Now, one of the necessary shortcomings of this AbstractFactory Designing pattern is we have to create A ConcreteFactory for each of the products. The Prototype-based approach eliminates the need for a new concrete factory class for each new product family.** We will discuss it later.

**Java Example Of AbstractFactory Designing Patterns:**

**/\*abstract base product class\*/**public abstract class Computer

{

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

//we make these methods abstract so that the abstract class does not need to define them. And, the normal class extends abstract class must define them

@Override

public String toString()

{

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

/\*Concrete product class\*/

public class PC extends Computer

{

private String ram;

private String hdd;

private String cpu;

public PC(String ram, String hdd, String cpu)

{

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM()

{

return this.ram;

}

@Override

public String getHDD()

{

return this.hdd;

}

/\*class Server\*/

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**/\*abstract factory method\*/**

public interface ComputerAbstractFactory

{

public Computer createComputer();

}

**/\*Concrete factory method\*/**

**/\*Now, we need to create Concrete Factory method for each of the prodducts\*/**

public class PCFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public PCFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new PC(ram,hdd,cpu);

}

}

/\*Another one\*/

public class ServerFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public ServerFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new Server(ram,hdd,cpu);

}

}

Now a consumer class will be created that will provide the entry point for the client classes to create sub-classes.

public class ComputerFactory {

public static Computer getComputer(ComputerAbstractFactory factory){

return factory.createComputer();

}

}

**/\*The designing pattern tester\*/**

public class TestDesignPatterns {

public static void main(String[] args) {

testAbstractFactory();

}

private static void testAbstractFactory() {

Computer pc = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2 GB","500 GB","2.4 GHz"));

Computer server = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new ServerFactory("16 GB","1 TB","2.9 GHz"));

System.out.println("AbstractFactory PC Config::"+pc);

System.out.println("AbstractFactory Server Config::"+server);

}

}

**What would be the equivalent Factory Design Pattern:**

**/\*A Base product class\*/**public abstract class Computer {

public abstract String getRAM();

public abstract String getHDD();

public abstract String getCPU();

@Override

public String toString(){

return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();

}

}

**/\*Now, this one is First ConcreteProduct Class\*/**

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**/\*Another ConcreteProduct class\*/**

public class Server extends Computer {

private String ram;

private String hdd;

private String cpu;

public Server(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public String getRAM() {

return this.ram;

}

@Override

public String getHDD() {

return this.hdd;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**/\*First, there is no AbstractFactory class\*/**

**/\*Factory Class\*/**

**/\*It will only produce one product as a time\*/**

public class ComputerFactory {

public static Computer getComputer(String type, String ram, String hdd, String cpu){

if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);

else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd, cpu);

return null;

}

}

**Abstract Factory Design Pattern Pros And Cons:**

**Pros:**

Follows the Open/Closed Principle.

Allows building families of product objects and guarantees their compatibility.

Avoids tight coupling between concrete products and code that uses them.

Divides responsibilities between multiple classes.

**Cons:**

Increases overall code complexity by creating multiple additional classes.

**The Difference Between Factory And AbstractFactory:**

Both factory and abstract factory are creational design patterns. The major difference between these two is, a factory pattern creates an object through inheritance and produces only one Product. On the other hand, an abstract factory pattern creates the object through composition and produce families of products.

**The Difference Between Builder And Factory:**

Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.

Builder often builds a Composite.

(Builder is used when the product cannot be produced in a single step)

(AbstractFactory can return a result contains a complex set of products. Builder generates a final product)