SourceMaking
teaching IT professionals

Log in | Contact

Design Patterns | Antipatterns | Refactoring | UML

- Design Patterns Reference
- Design Patterns Book

Home › Design Patterns › Creational patterns

Abstract Factory Design Pattern

Intent

01
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".

- The `new` operator considered harmful.

Problem

02 If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulatation is not engineered in advance, and lots of `#ifdef` case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

[+] Feedback

BOOKMARK

[Search] ✕

Discussion

Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The "factory" object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

Structure

The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the `new` operator and the concrete, platform-specific, product classes. Each "platform" is then modeled with a Factory derived class.
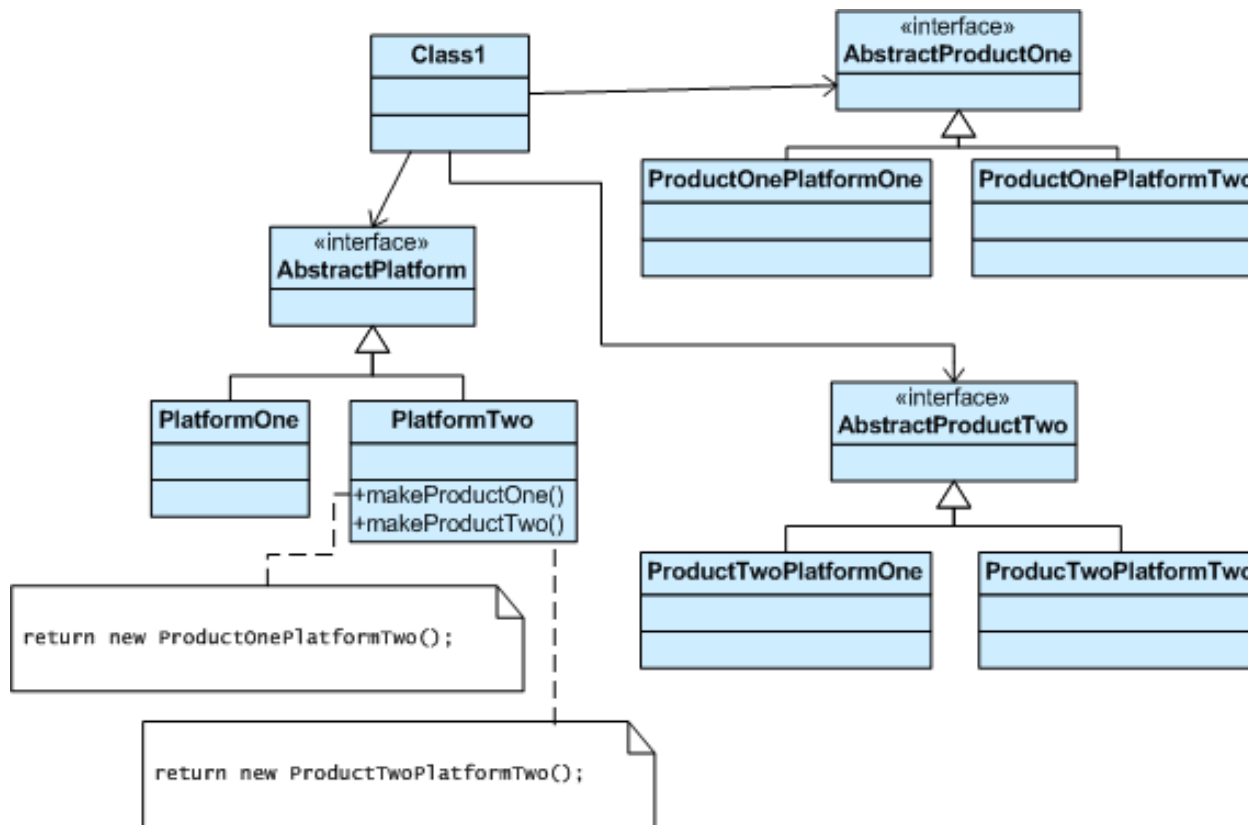
Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.
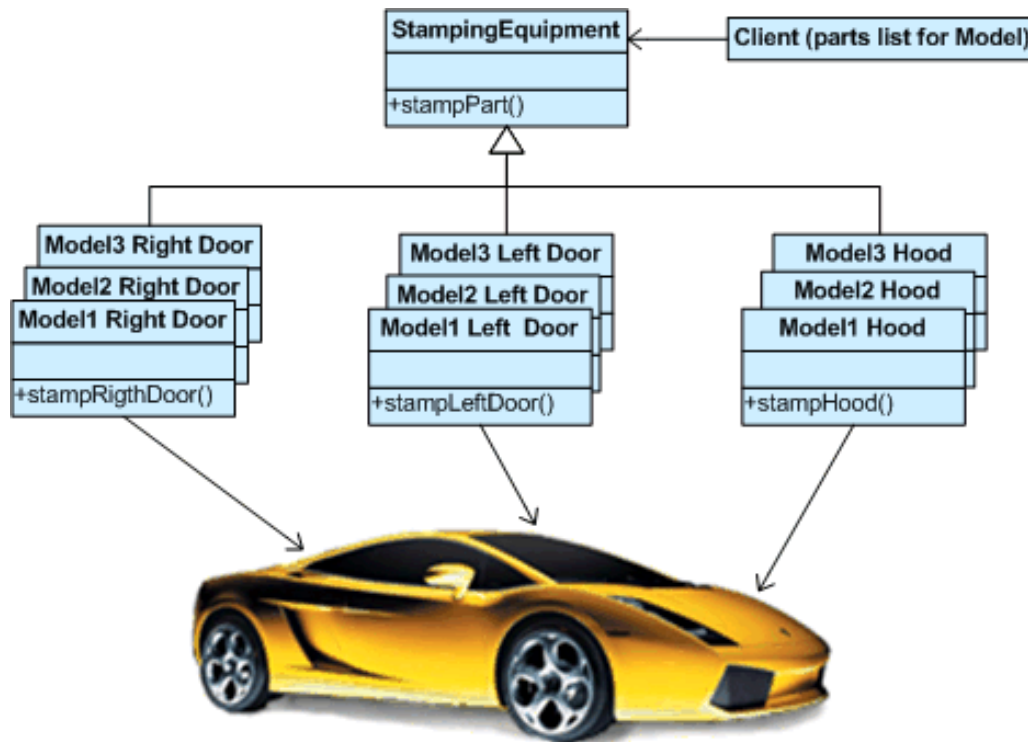
Check list

1. Decide if "platform independence" and creation services are the current source of pain.

2. Map out a matrix of "platforms" versus "products".

3. Define a factory interface that consists of a factory method per product.

4. Define a factory derived class for each platform that encapsulates all references to the `new` operator.

5. The client should retire all references to `new`, and use the factory methods to create the product objects.

Rules of thumb

- Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects, Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation.

- Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object.

- Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.

- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.

- 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.

- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Read next
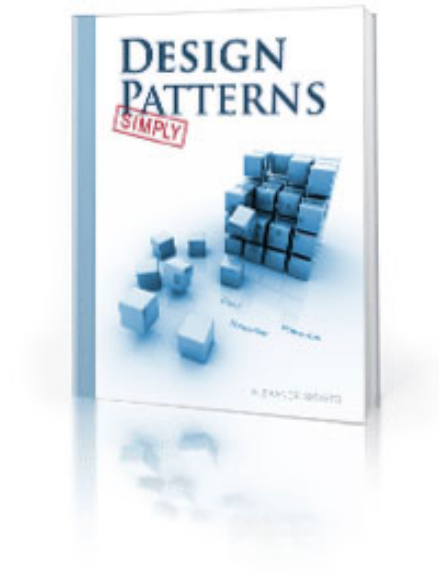
This article is taken from the book **Design Patterns - Simply**.

ALL design patterns are compiled there. The book is written in a clear and simple language that makes it easy to read and understand (just like this article).

It is a part of our Design Patterns Course. We distribute it in PDF format, so it will be available for downloading in 10 seconds!

[Learn more...](#)

Abstract Factory code examples

C# examples

- [Abstract Factory in C#](#)

C++ examples

- [Abstract Factory in C++: Before and after](#)
- [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 | Builder Design Pattern ›