

## More Patterns

Design Patterns: Elements of  
Reusable Object-Oriented  
Software  
by Gamma, Helm, Johnson,  
Vlissides,  
GoF95



## Creational Patterns

- These patterns provide guidance as to how to create objects when their creation requires a decision.
  - Dynamical instantiation of objects
  - How to structure and encapsulate these decisions
- Examples include Factory, Abstract Factory, Builder, Prototype, Singleton, Object Pool.



## Factory – A Creational Pattern

GOF95 (p. 107)  
Grand98 (p. 89)  
Patterns in Java, vol. 1 by Mark  
Grand, 1998



### The Pattern

- When you write a class to be reusable with arbitrary data types, it can be useful to organize it so that when it wants to instantiate a class, it delegates the choice of which class to instantiate to another object.
- The Factory Method pattern provides a way to do this.



## Related Pattern: Abstract Factory



- The Factory Method pattern is useful for constructing individual objects for a specific purpose without the construction requester knowing the specific classes being instantiated.
- If you need to create a matched set of such objects, then the Abstract Factory pattern is a more appropriate pattern. [Grand98]

## Intent (*GoF95, pp. 107-116*)



- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.

## Synopsis (*Grand98, pp. 89-98*)



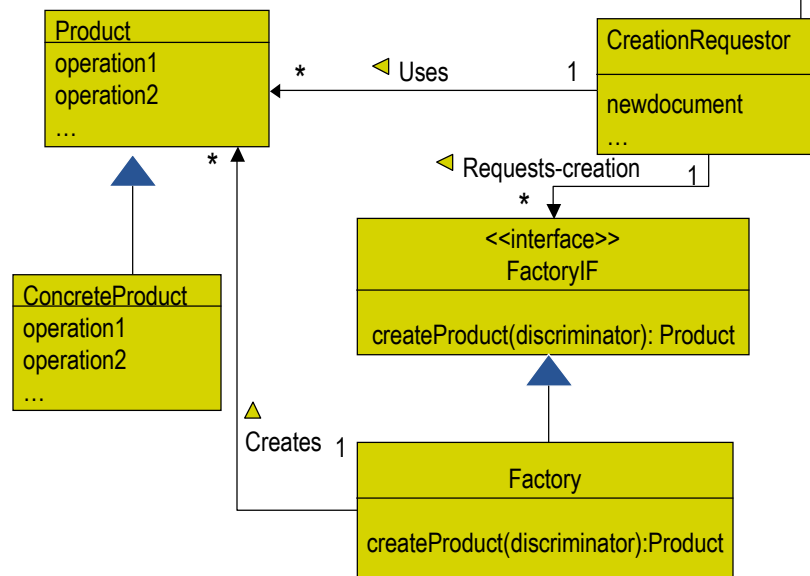
- You write a class for reuse with arbitrary data types.
- You organize this class so that it can instantiate other classes without being dependent on any of the classes it instantiates.

## Synopsis (*Grand98, pp. 89-98*)



- The reusable class is able to remain independent of the classes it instantiates by delegating the choice of which class to instantiate to another object and referring to the newly created object through a common interface.

## Factory Pattern



## 5 Classes or Interfaces

```
public class CreationRequestor
    extends Object
```

- A method of `CreationRequestor` requests a `ConcreteProduct` instance from a `Factory`.
- `CreationRequestor` only knows the abstract class `Product` (extended by `ConcreteProduct`) and the interface `FactoryIF` (implemented by `Factory`).

## 5 Classes



```
public interface FactoryIF
```

- Provides a method that returns a new object of type Product.

## 5 Classes



```
public class Factory
    extends Object
        implements FactoryIF
```

- Defines the method that returns a new object of type Product.
- This method instantiates an object of the class ConcreteProduct.

## 5 Classes



```
public abstract class Product
    extends Object
```

- Provides an abstract method which we will name `doIt`.

## 5 Classes



```
public class ConcreteProduct
    extends Product
```

- Effectively implements the `doIt` method.

## Source Code of Grand's Classes



```
public class CreationRequestor extends Object {
    private final FactoryIF factory;
    public CreationRequestor(FactoryIF factory_1) {
        this.factory = factory_1 ;
        /*...*/
    }
    public void someMethod() {
        Product product = this.factory.newProduct() ;
        product.doIt() ;
    }
}
```

## Source Code of Grand's Classes



```
public interface FactoryIF {
    public abstract Product newProduct() ;
}

public class Factory
    extends Object
    implements FactoryIF {
    public Product newProduct() {
        return new Product() ;
    }
}
```

## Source Code of Grand's Classes



```
public abstract class Product
    extends Object {
    public abstract void doIt() ;
}

public class ConcreteProduct
    extends Product {
    public ConcreteProduct() {
        /*...*/
    }
    public void doIt() {
        /*...*/
    }
}
```

## Abstract Factory - Another Creational Pattern

GOF95 p. 87  
Grand98 p. 99



## Intent

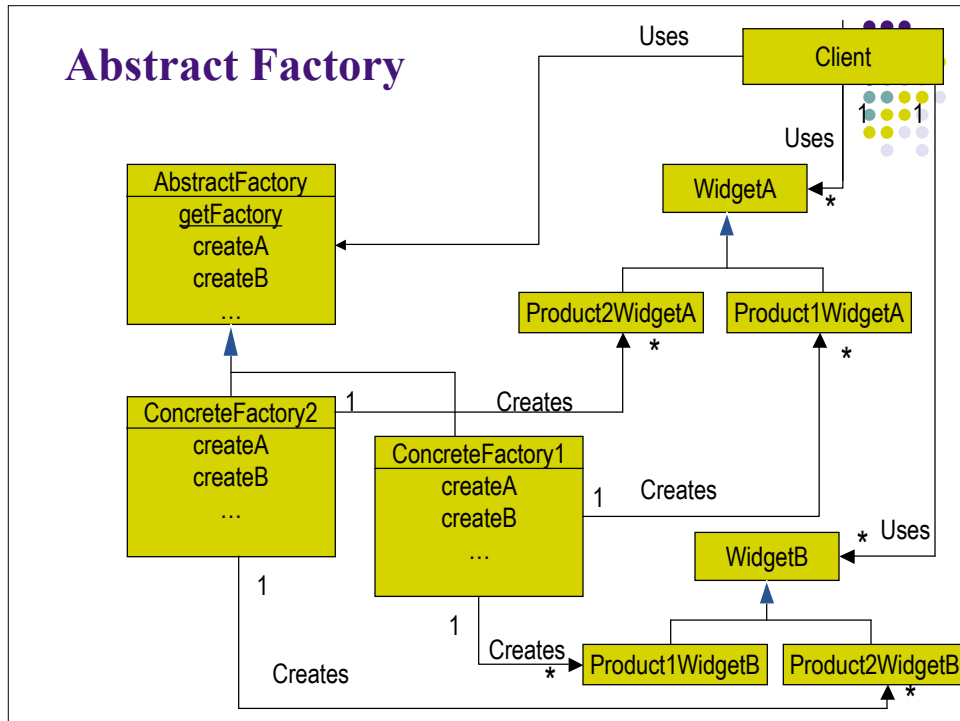


- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

## Problem



- 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 encapsulation is not engineered in advance, and lots of `#ifdef` case statements with options for all currently supported platforms begin to appear throughout the code.



## Client

- Client classes use various widget classes to request or receive services from the product that the client is working with.
- Client classes only know about about the abstract widget classes.
- They have no knowledge of the concrete widget classes.

## AbstractFactory



- These classes define abstract methods for creating instances of concrete widget classes.
- They have a static method – `getFactory` or `getToolkit`.
- A client object calls that method to get an instance of a concrete factory appropriate for creating widgets that work with a particular product.

## ConcreteFactory1/2



- These classes implement the methods defined by their abstract factory superclasses to create instances of concrete widget classes.
- Client classes that call these methods do not have any direct knowledge of the concrete factory classes.

## WidgetA/B



- These are abstract classes that correspond to a feature of a product that their concrete subclasses will work with.
- These are abstract widgets.

## Product1WidgetA, etc.



- Concrete classes that correspond to a feature of a product that they work with.
- These are the concrete widgets.

## Discussion



- Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes.

## Discussion



- 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 it for them.

## Discussion



- 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 replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

## Discussion



- Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.
  - The Singleton pattern ensures that only one instance of a class is created.
  - All objects that use an instance of that class use the same instance.

## Example



- The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes.

## The Sheet Metal Stamper



- The Abstract Factory pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles.
  - Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p. 54

## The Sheet Metal Stamper



- 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.

## Rules of thumb



- Sometimes creational patterns are competitors.
  - There are cases when either Prototype or Abstract Factory could be used profitably.

## Prototype Pattern



- This pattern allows an object to create customized objects without knowing their exact class or the details of how to create them.
  - It gives prototypical objects to the object that initiates the creation of objects.
  - The creation-initiating object then creates objects by asking the prototypical objects to make copies of themselves.

## Rules of thumb



- At other times they are complementary.
  - Abstract Factory might store a set of Prototypes from which to clone and return product objects [GOF, p126].
  - 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. [GOF, pp81,134]

## Builder Pattern



- The Builder pattern allows a client object to construct a complex object by specifying only its type and content.
- The client is shielded from the details of the object's construction.

## Rules of thumb



- Abstract Factory, Builder, and Prototype define a factory object that is 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.



## Rules of thumb

- 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. [GOF, p135]



## Rules of thumb

- Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype. [GOF, p95]
- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

## Façade Pattern



- The façade pattern simplifies access to a related set of objects by providing one object that all objects outside the set use to communicate with the set.

## Rules of thumb



- 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. [GOF, p105]

## Rules of thumb



- 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. [GOF, p136]

Object Pool

(Grand98, p. 135)



## Synopsis



- Manage the reuse of objects for a type of object that is expensive to create or only a limited number of a kind of object can be created.

## Related Patterns



- Cache Management (Structural Pattern)
  - The Cache Management pattern manages the reuse of specific instances of a class.
  - The Object Pool pattern manages and creates instances of a class that can be used interchangeably. [Grand98]

## Related Patterns



- **Factory Method**

- The Factory Method pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation. [Grand98]

- **Singleton**

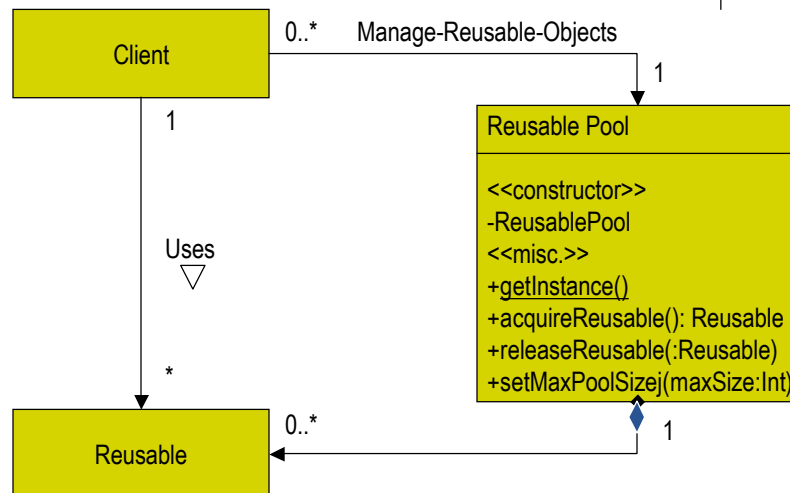
- Objects that manage object pools are usually singletons. [Grand98]

## Context



- A program may not create more than a limited number of instances of a particular class.
- If creating instances is very expensive then this activity should be avoided.
- By reusing objects when it is finished with them, a program can avoid creating new objects.

## Object Pool



## Object Pool

- Reusable
  - Instances of this class collaborate with other objects for a limited time, then they are no longer needed.
- Client
  - Use Reusable objects.
- ReusablePool
  - Manage Reusable objects for use by Client objects.



## Object Pool

- It is desirable to keep all `Reusable` objects that are not currently in use in the same object pool.
  - Managed by one coherent policy.
- The `ReusablePool` class is a `Singleton` class.
  - Its constructors are private.
  - Other classes must call its `getInstance` method to get an instance of the `ReusablePool` class.



## Object Pool

- When a `Client` object needs a `Reusable` object,
  - Call a `ReusablePool` object's `acquireReusable` method
- A `ReusablePool` object maintains a collection of `Reusable` objects.
  - Contains objects that are not in use.



## Object Pool

- If there are unused `Reusable` objects in the pool, one is removed and returned.
- If the pool is empty, `acquireReusable` will create a `Reusable` object if it can.
  - If it cannot, then it waits until a `Reusable` object is returned to the pool.



## Object Pool

- When a `Client` object is finished with the `Reusable` object, it passes it to a `ReusablePool` object's `releaseReusable` method which returns the `Reusable` object to the pool of not in use `Reusable` objects.