**CREATIONAL DESIGN PATTERNS**

* **Creational design patterns** abstract the instantiation process.
* They help make a system independent of how its objects are created, composed, and represented.
* A **class creational pattern** uses inheritance to vary the class that's instantiated, whereas an **object creational pattern** will delegate instantiation to another object.
* Creational patterns become important as systems evolve to depend more on object composition than class inheritance.
* Consequently, the creational patterns give a lot of flexibility in **what** gets created, **who** creates it, **how** it gets created, and **when**.
* Sometimes creational patterns are competitors.

**Abstract Factory Design Pattern**

**DEFINATION:-**

Providing an interface for creating the families if related objects/

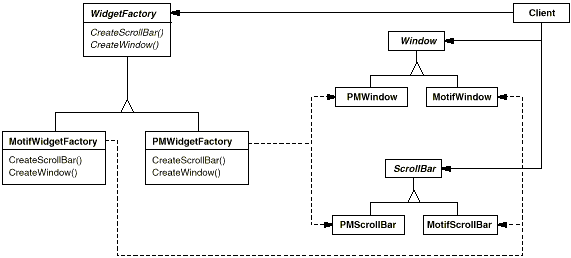
**Intent:**

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

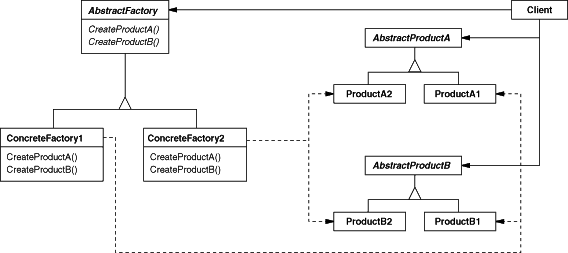
**Also Known As: Kit**.

**Motivation:**

* Consider a user interface toolkit that supports multiple look-and-feel standards. Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons.
* We can solve this problem by defining an abstract Widget Factory 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.
* WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class.

**Applicability:**

1. Use the Abstract Factory pattern when a system should be independent of how its products are created, composed, and represented.
2. A system should be configured with one of multiple families of products.
3. A family of related product objects is designed to be used together, and you need to enforce this constraint.
4. You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
5. **Structure:**

****

1. **Participants:**
2. **AbstractFactory (WidgetFactory)**

-> declares an interface for operations that create abstract product objects.

2. **AbstractProduct (Window, ScrollBar)**

**->** declares an interface for a type of product object.

**3. ConcreteProduct (MotifWindow, MotifScrollBar)**

🡪Defines a product object to be created by the corresponding concrete factory.

🡪Implements the Abstract Product interface.

4. **Client:** uses only interfaces declared by Abstract Factory and Abstract Product classes.

**Collaborations**:

* Normally a single instance of a Concrete Factory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
* Abstract Factory defers creation of product objects to its Concrete Factory subclass.

**Consequences:**

The Abstract Factory pattern has the following benefits and liabilities:

1. **It isolates concrete classes***. The Abstract Factory pattern helps you control the classes of objects* that an application creates.
2. **It makes exchanging product families easy***.*
3. **It promotes consistency among products.**
4. **Supporting new kinds of products is difficult.**

**Implementation:**

* Here are some useful techniques for implementing the Abstract Factory pattern.

1. *Factories as singletons. An application typically needs only one instance of a Concrete Factory* per product family. So it's usually best implemented as a Singleton.

2. *Creating the products. Abstract Factory only declares an interface for creating products.*

**Sample Code:**

**Related Patterns:**

* Abstract Factory classes are often implemented with factory methods , but they can also be implemented using Prototype.

A concrete factory is often a singleton.

**Builder**

**DEFINATION:-**

“Builder pattern builds a complex object using simple objects and using a step by step approach”.

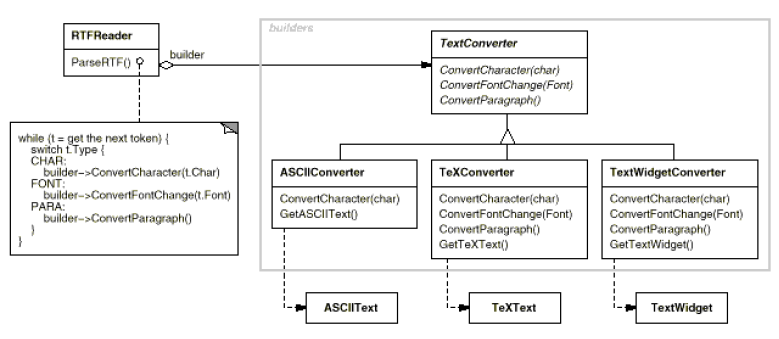
The process of constructing a complex object should be generic so that the same process can be used to create different representations of the same complex object.

**Intent:**

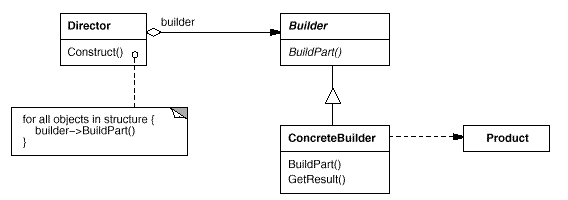
* Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Motivation:**

* A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats.
* The reader might convert RTF documents into plain ASCII text that can be edited interactively.
* So it should be easy to add a new conversion without modifying the reader.
* A solution is to configure the RTF Reader class with a Text Converter object that converts RTF to another textual representation. As the RTF Reader parses the RTF document, it uses the Text Converter to perform the conversion.
* Whenever the RTF Reader recognizes an RTF token it issues a request to the Text Converter to convert the token.
* Text Converter objects are responsible both for performing the data conversion.



* The converter is separate from the reader, which is responsible for parsing an RTF document.
* The Builder pattern captures all these relationships.
* Each converter class is called a **builder** in the pattern**,** and thereader is called the **director.**

**Structure: **

**Applicability:**

* Use the Builder pattern:

1. When the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
2. The construction process must allow different representations for the object that's constructed.

**Participants:**

1. **Builder (TextConverter):** specifies an abstract interface for creating parts of a Product object.
2. **Concrete Builder (ASCII Converter, Text Converter, TextWidgetConverter):** constructs and assembles parts of the product by implementing the Builder interface.

* defines and keeps track of the representation it creates.

**3. Director (RTFReader):** constructs an object using the Builder interface.

**4. Product (ASCIIText, TeXText, TextWidget).**

**Collaborations:**

* The client creates the Director object and configures it with the desired Builder object.
* Director notifies the builder whenever a part of the product should be built.
* Builder handles requests from the director and adds parts to the product.
* The client retrieves the product from the builder.
* The following interaction diagram illustrates how Builder and Director cooperate with a client.

**Consequences:**

1. It lets you vary a product's internal representation.

2. It isolates code for construction and representation.

* The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented.
* Clients needn't know anything about the classes that define the product's internal structure.

1. It gives you finer control over the construction process.

**Implementation:-**

Assembly and construction interface:-

Builders construct their products in step by step fashion.

The builder class interface must be general enough to allow the construction of products for all kinds of concrete builders.

**Known Uses:**

* The RTF converter application is from ET++ .
* Builder is a common pattern in Smalltalk-80.
* ClassBuilder is a builder that Classes use to create subclasses for themselves. In this case a Class is both the Director and the Product.

**Related Patterns:**

* **Abstract Factory** is similar to Builder in that it too may construct complex objects.
* The primary difference is that the Builder pattern focuses on constructing a complex object step by step. Abstract Factory's emphasis is on families of product objects (either simple or complex).
* Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.
* A Composite is what the builder often builds.

**Factory Method**

**Defination:-**

In factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

In simple words, if we have a super class and n sub classes and based on the data provided, we have to return the object of one of the sub classes, we use a factory pattern.

The basic principle behind this pattern is that, at run time we get an object of similar type based on the parameter we pass.

If the object creation code is spread in whole application, and if you need to change the process of object creation then you need to go in each and every place to make necessary changes.

**Intent:**

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

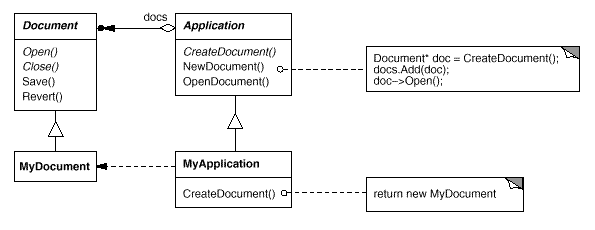
**Also Known As:**

* Virtual Constructor.

**Motivation:**

* To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument.
* The Application class is responsible for managing Documents and will create them as required—when the user selects Open or New from a menu, for example.

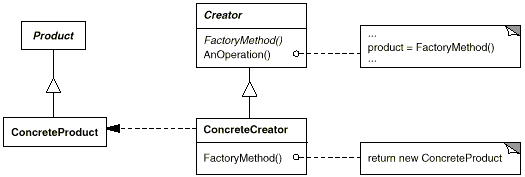
Because the particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate—the Application class only knows *when a new document should be created,* not *what kind of Document to create.*



* We call CreateDocument a **factory method because it's responsible for** "manufacturing" an object.

**Applicability:**

* Use the Factory Method pattern when a class can't anticipate the class of objects it must create.
* A class wants its subclasses to specify the objects it creates.

**Structure:** 

**Participants:**

1. **Product (Document):** Defines the interface of objects the factory method creates.
2. **ConcreteProduct (MyDocument):** implements the Product interface.
3. **Creator (Application):** Declares the factory method, which returns an object of type Product.
4. **ConcreteCreator (MyApplication):** Overrides the factory method to return an instance of a ConcreteProduct.

**Collaborations:**

* Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.

**Consequences:**

* Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined Concrete Product classes.
* Provides hooks for sub classes
* Creating objects inside a class with a factory method is always more flexible than creating an object directly.
* It gives subclasses a hook for providing an extended version of an object.

**Known Uses:**

* Factory methods pervade toolkits and frameworks.

**Related Patterns:**

* **Abstract Factory** is often implemented with factory methods.
* Factory methods are usually called within **Template Methods.**
* **Prototypes** don't require subclassing Creator.
* Creator uses Initialize to initialize the object.
* Factory Method doesn't require such an operation.

**Prototype**

**Defination:-**

The proto type design pattern is used for creating new objects (instances) by cloning (copying) other objects and this way we can improve the performance. This pattern uses java cloning to copy the object.

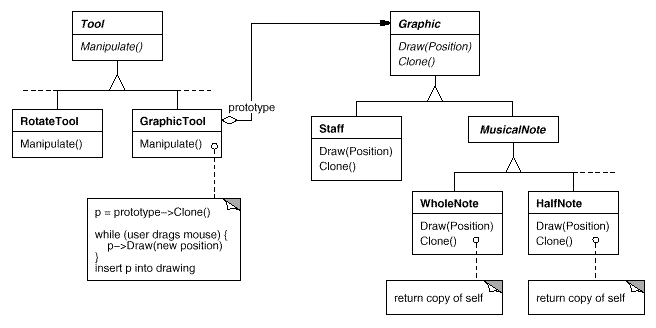
This pattern is used when creation of object is costly or complex. For example, a object is to created after a costly database operation. We can cache the object, returns its clone on next request. Once we get the cloned object we can modify according to our needs.

**Intent:**

* Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

**Motivation:**

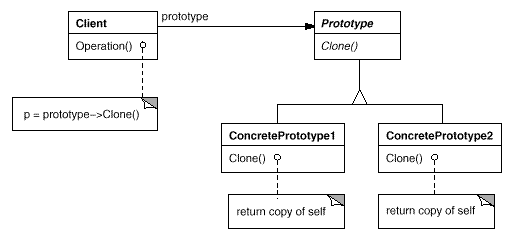
* How can the framework use it to parameterize instances of GraphicTool by the class of Graphic they're supposed to create?
* The solution lies in making GraphicTool create a new Graphic by copying or "cloning" an instance of a Graphic subclass.
* We call this instance a **prototype.**



**Applicability:**

* Use the Prototype pattern when a system should be independent of how its products are **created, composed**, and **represented**.
* When the classes to instantiate are specified at run-time, it may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

**Structure:**



**Participants:**

1. **Prototype (Graphic)**

* declares an interface for cloning itself.

**2. ConcretePrototype (Staff, WholeNote, HalfNote)**

* implements an operation for cloning itself.

**3. Client (GraphicTool)**

* creates a new object by asking a prototype to clone itself.

**Collaborations:**

* A client asks a prototype to clone itself.

**Consequences:-**

Prototype has many of the same consequences that abstract factory and builder have.

1. Adding and removing products at runtime.
2. Specifying new objects by varying values.
3. Specifying new objects by varying structure.
4. Reduced sub classing.
5. Configuring an application with classes dynamically.

**Related Patterns:**

* **Prototype** and **Abstract Factory** are competing patterns in some ways.
* They can also be used together, however, An Abstract Factory might store a set of prototypes from which to clone and return product objects.
* Designs that make heavy use of the **Composite** and **Decorator patterns** often can benefit from Prototype as well.

**Singleton**

**Definition:-**

Singleton pattern is a design solution where an application wants to have one and only one instance of the class.

We can access the singleton instance globally.

Singleton pattern will ensure that there is only one instance of a class is created.

**Intent:**

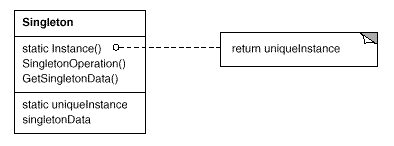
* Ensure a class only has one instance, and provide a global point of access to it.

**Motivation:**

* It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager.
* A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.
* A better solution is to make the class itself responsible for keeping track of its sole instance.
* The class can ensure that no other instance can be created and it can provide a way to access the instance.
* This is the **Singleton pattern**.

**Applicability:**

* Use the Singleton pattern when there must be exactly one instance of a class, and it must be accessible to clients from a well known access point.
* When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

**Structure:**

**Participants:**

1. **Singleton**

* Defines an Instance operation that lets clients access its unique instance.

**Collaborations:**

* Clients access a Singleton instance solely through Singleton's Instance operation.

**Consequences:**

* The Singleton pattern has several benefits as following:

1. Controlled access to sole instance.
2. Reduced name space.
3. Permits refinement of operations and representation.
4. Permits a variable number of instances.
5. More flexible than class operations.

**Implementation:-**

Ensuring a unique instance.

The singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created.

**Related Patterns:**

* Many patterns can be implemented using the Singleton pattern. For example in Abstract Factory, Builder, and Prototype.