Creational Patterns

Introduction

Creational patterns deals with object creation process. In object oriented software development, while designing and implementation, designers and coders work with classes and objects. Different types of problems might require different object interactions to solve a particular problem. Creating objects as per the problem requirements is an important design issue. The creational patterns deal with this issue and help the designers to solve and guide the object creation mechanisms.

Creational design patterns are further classified into: class-creational patterns and object-creational patterns. Class-creational patterns deals with instantiation of a subclass based on the context. Object-creational patterns deals with delegation of object creation process to another object.

There are five creational patterns:

1) Abstract Factory
2) Builder
3) Factory Method
4) Prototype
5) Singleton

Singleton Pattern

Intent:
To create exactly one object of a class, and to provide global point of access to it.

Motivation:
While developing applications, we will face frequent situations where there is a need to create exactly one instance of a class. For example, in a GUI application there is a requirement that there should be only one active instance of the application's user interface. No other instance of the user interface of the application should be created by the user. Another example is the requirement to maintain only one print spooler.

We need to allow the class to have exactly one instance and to allow a global point of access to it. How to do it? A global variable might allow you to provide global access to the object but it does not restrict you from creating multiple objects.

A better solution is to allow the class to take the responsibility of creating only one object and to provide a global point of access to it. This is what the singleton pattern does. The singleton pattern allows the class to create only one instance, allow other external objects to access that instance and does not allow the constructor to be invoked directly each time.

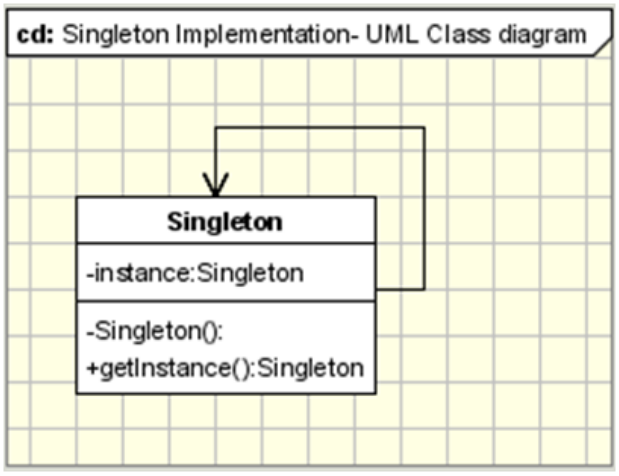
Applicability:
Singleton Pattern can be used when:

1) There must be exactly one instance of a class and to provide a global point of access to it.
2) When the sole instance should be extensible by sub classing, and clients should be able to use the extended instance without modifying its code.

Singleton pattern can be used for developing in following situations:

- Logger classes
- Configuration classes
- Accessing resources in shared mode
- Factories implemented as Singletons

Structure:



Participants:
Consists of a "Singleton" class with the following members:

1) A static data member
2) A private constructor
3) A static public method that returns a reference to the static data member. This method is responsible for creating the single instance and providing the global access to that instance.

Collaborations:
Clients access the singleton instance solely through the singleton's class operation.

Consequences:

1) Controlled access to sole instance: As the instance is completely encapsulated in the class, we can have strict control on how the clients access it.
2) Reduced name space: Avoids polluting the name space with global variables that store the single instances.
3) Permits refinement of operations and representation: It easy to subclass and extend the functionality of the class method.
4) Permits a variable number of instances: It is possible to create multiple instances by changing the code in the class method.
5) More flexible than class operations: In C++, class methods cannot be overriden. Instance methods provides the flexibility of extending the operation in the subclass.

Implementation:
The Singleton pattern defines a getInstace() class method which creates the new instance and is responsible for maintaining a single instance of the class.
It also provides global access to the clients as it will be a public method. The implementation code will be as follows:

```
class Singleton
{
        private static Singleton instance;
        private Singleton()
        {
                ...
        }
        public static synchronized Singleton getInstance()
        {
                if (instance == null)
                        instance = new Singleton();
                return instance;
        }
        ...
        public void doOperation()
        {
                ...
        }
}
```

We can see from the above code that the getInstance() method ensures that only one instance of the class is created. The constructor must not be accessible from outside the class. So, it is made

private. The only way for instantiating the class is through getInstance() method. It is also responsible for providing a global point of access to the instance as shown below:
Singleton.getInstance().doSomething();

Specific problems and implementations:

Thread-safe implementation for multi-threading environment:
In a multithread application, multiple threads will be performing operations on a Singleton instance. To avoid undesired results, the Singleton class should provide a way to solve this problem. As shown in the above code, the getInstance() is marked as synchronized to avoid read/write problems on the Singleton instance by multiple threads.

Lazy instantiation using double locking mechanism:
The implementation code provided above is thread-safe but it is not the best thread-safe implementation of Singleton. We can observe in the above code that when ever a new instance is created, in a multithreaded application, synchronization mechanisms are invoked as the method getInstance() is marked as synchronized. So, this implementation code is not best in the performance dimension. To improve the performance, we should modify the above code in such a way that synchronization mechanism is invoked only when the instance is created for the first time. This performance optimization involves a check whether an instance already exists in a non-synchronized block or not. If there is already an instance, return it. If not, a new instance is created inside a synchronized block after performing another check. This process is known as double locking mechanism. Now, after implementing this strategy, the code will look as shown below:

```
//Lazy instantiation using double locking mechanism.
class Singleton
{
	private static Singleton instance;
	private Singleton()
	{
		System.out.println("Singleton(): Initializing Instance");
	}
	public static Singleton getInstance()
	{
		if (instance == null)
		{
			synchronized(Singleton.class)
			{
				if (instance == null)
				{
					System.out.println("getInstance(): First time getInstance was invoked!");
					instance = new Singleton();
				}
			}
		}
		return instance;
	}
	public void doSomething()
	{
		System.out.println("doSomething(): Singleton does something!");
```

```
        }
}
```

This double checking method is not supported in previous versions before java 5. For a universal solution, we can use the nested class approach.

Nested class solution by Bill Pugh:

This technique is known as **initialization on demand holder.** In this, a nested class is used to create the instance. This solution is thread-safe without requiring special language constructs like volatile and synchronized. Since the nested class member will be final, the instance will be created only on its first reference. The code for this solution will be as shown below:

```
public class Singleton
{
    // Private constructor prevents instantiation from other classes
    private Singleton() { }

    /**
    * SingletonHolder is loaded on the first execution of Singleton.getInstance()
    * or the first access to SingletonHolder.INSTANCE, not before.
    */
    private static class SingletonHolder
    {
       public static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance()
    {
                        return SingletonHolder.INSTANCE;
    }
}
```

Early instantiation implementation using a static field:

In this Singleton's implementation, instance of the class is created when the class is loaded by the class loader as the data member will be marked as static. In this implementation, we don't need any synchronization mechanisms. As the class is loaded only once, this will guarantee that the instance created is unique. Singleton's implementation code using the early instantiation method is as follows:

```
//Early instantiation using implementation with static field.
class Singleton
{
        private static Singleton instance = new Singleton();
        private Singleton()
        {
                System.out.println("Singleton(): Initializing Instance");
        }
        public static Singleton getInstance()
        {
                return instance;
```

```
        }
        public void doSomething()
        {
                System.out.println("doSomething(): Singleton does something!");
        }
}
```

There is a problem with this implementation which is a rare case. The problem is when a Singleton class in a package is loaded by two different class loaders.

Sub classing (Protected Constructor):
We can allow the Singleton class to be extended by allowing creation of sub classes to this class. This can be possible when the private constructor in the Singleton class is made protected. This has two drawbacks:

1) If the constructor is made protected, another class in the same package can access it to create a new object. This defeats the purpose of a Singleton.
2) In the derived class or subclass, all the calls to the getInstance() method i.e., Singleton.getInstance() should be replaced with NewSingleton.getInstance().

Serialization:
If the Singleton class implements the java.io.Serializable interface, when a singleton is serialized and then deserialized more than once, there will be multiple instances of Singleton created. In order to avoid this the readResolve method should be implemented. The code will be as follows:

```
public class Singleton implements Serializable
{
        ...
        // This method is called immediately after an object of this class is deserialized.
        // This method returns the singleton instance.
        protected Object readResolve()
        {
                return getInstance();
        }
}
```

Sample Code:
Let us take a small practical example to understand the Singleton design pattern in more details.

Problem Statement:
Design a small ATM printing application which can generate multiple types of statements of the transaction including Mini Statement, Detailed statement etc. However the customer should be aware of the creation of these statements. Ensure that the memory consumption is minimized.

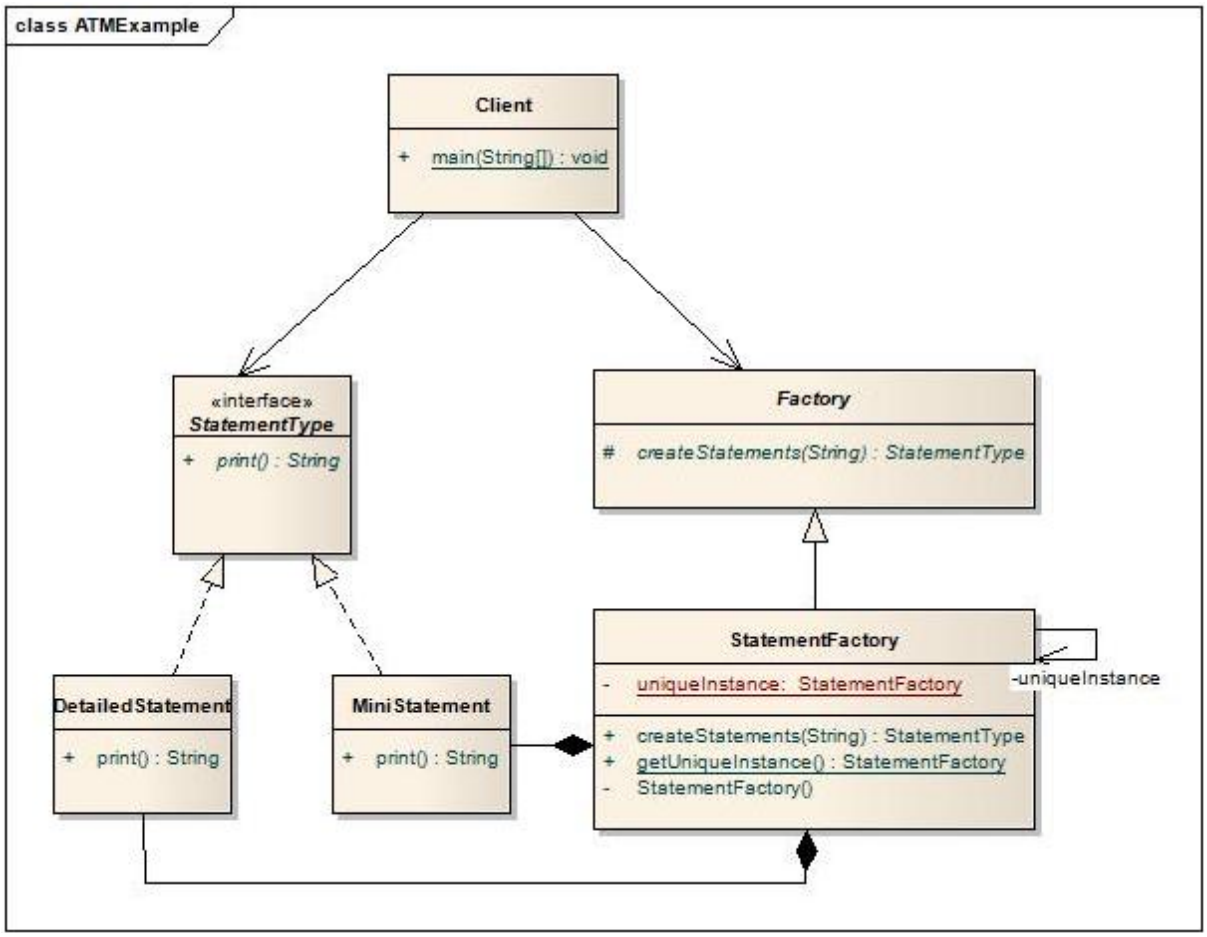
Design Solution:
The above requirement can be addressed using two core Gang of four design pattern – Factory design pattern and Singleton design pattern. In order to generate multiple types of statements for the ATM transactions in the ATM machine we can create a Statement Factory object which will have a factory method of *createStatements()*. The createStatements will create DetailedStatement or MiniStatement objects.

The client object will be completely unware of the object creation since it will interact with the Factory interface only. We will also create an interface called StatementType. This will allow further

statement type objects e.g. Credit card statement etc to be added. So the solution is scalable and extensible following the object oriented Open/Closed design principle.
The second requirement of reducing the memory consumption can be achieved by using Singleton design pattern. The Statement Factory class need not be initiated multiple times and a single factory can create multiple statement objects. Singleton pattern will create a single instance of the StatementFactory class thus saving memory.

Code for StatementFactory class is as shown below:

```
public class StatementFactory extends Factory
{
        private static StatementFactory uniqueInstance;

        private StatementFactory() {}

        public static StatementFactory getUniqueInstance()
        {
                if (uniqueInstance == null)
                {
                        uniqueInstance = new StatementFactory();
                        System.out.println("Creating a new StatementFactory instance");
                }
                return uniqueInstance;
        }

        public StatementType createStatements(String selection)
        {
                if (selection.equalsIgnoreCase("detailedStmt"))
                {
                   return new DetailedStatement();
                }
                else if (selection.equalsIgnoreCase("miniStmt"))
                {
                      return new MiniStatement();
                }
                throw new IllegalArgumentException("Selection doesnot exist");
         }
}
```

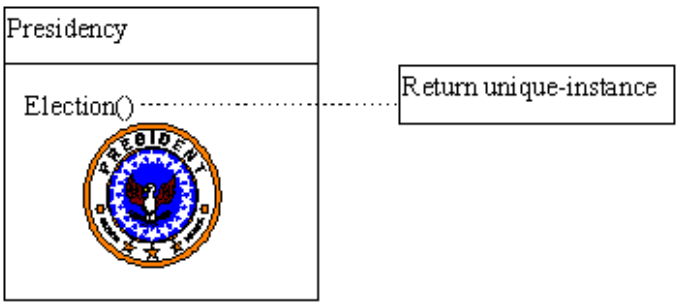


Known Uses:
Following are the example usage of Singleton pattern in real world:

1) Singleton pattern in Samlltalk-80, is the set of changes to the code, which is ChangeSet current.
2) Relationship between classes and their metaclasses.
3) InterViews user interface toolkit in Smalltalk uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes.

Related Patterns:
Other patterns like Abstract Factory, Builder and Prototype can be implemented using the Singleton pattern.
Example Singleton classes in java API:
java.lang.Runtime
java.awt.Desktop

Non-software example:
The *Singleton* pattern ensures that a class has only one instance, and provides a global point of access to that instance. The *Singleton* pattern is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a *Singleton*. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.


Presidency
Election()
Return unique-instance

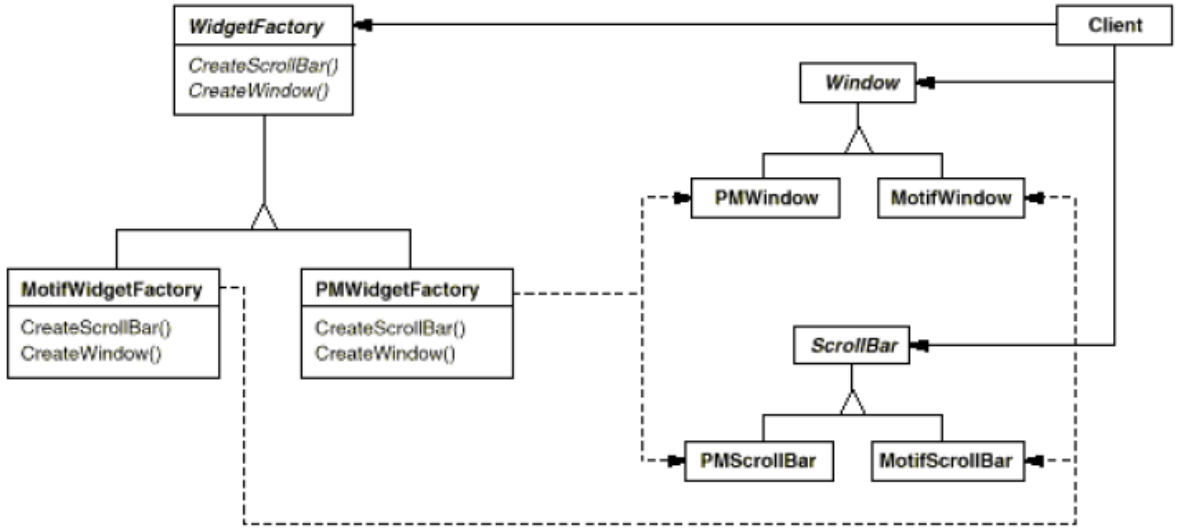
Abstract Factory Pattern

Intent:
Abstract factory provides an interface to create a family of related objects, without explicitly specifying their concrete classes.

Also Known As:
Kit.

Motivation:
Modularization is a big issue in today's programming. Programmers are trying to avoid the idea of adding more code to the existing classes to encapsulate more general information. Let's consider an example of user interface toolkit which 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 scrollbars, buttons, radio buttons etc… To be portable across different look-and-feels standards, an application should not hard code its widgets to a particular look-and-feel. It becomes difficult to change the look-and-feel at a later stage if we did so.

This problem can be solved by creating an abstract WidgetFactory class that declares an interface for creating a particular widget. There's also an abstract class for every kind of widget whose subclasses provide the implementation for creating a widget of certain look-and-feel. Clients call the operations in the interface provided by WidgetFactory which returns an instance of the abstract widget class. Clients aren't aware of the concrete classes they are using. Thus clients stay independent of the look-and-feel.



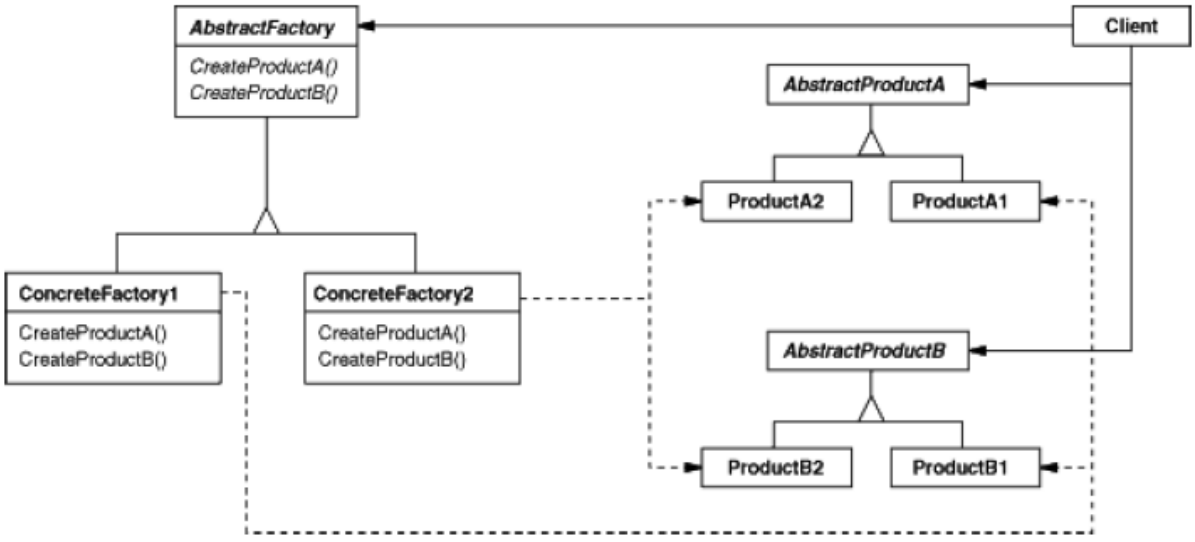
Applicability:
We should use Abstract Factory design pattern when:

1) The system needs to be independent of the products it works with are created.
2) The system should be configured to work with multiple families of products.
3) A family of products is designed to be used together and this constraint is needed to be enforced.
4) A library of products is to be provided and only their interfaces are to be revealed but not their implementations.

Examples where the Abstract Factory pattern can be used:

- Phone numbers
- Pizza factory
- Look & feel

Structure:



Participants:

The classes that participate in the Abstract Factory are:

AbstractFactory: Declares an interface of operations that create abstract products.

ConcreteFactory: Implements operations to create concrete products.

AbstractProduct: Declares an interface for a type of product objects.

Product: Defines a product to be created by the corresponding ConcreteFactory. It implements the AbstractProduct interface.

Client: Uses the interfaces declared by the AbstractFactory and AbstractProduct classes.

Collaborations:

- The ConcreteFactory class creates products objects having a particular implementation. To create different product, the client should use a different concrete factory.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Consequences:

Following are the benefits and liabilities of Abstract Factory pattern:

1. It isolates concrete classes.
2. It makes exchanging product families easy.
3. It creates consistency among products.
4. Supporting new kinds of products is difficult.

Implementation:

The AbstractFactory class is the one that determines the actual type of the concrete object and creates it, but it returns an abstract reference to the concrete object just created. This determines the behavior of the client that asks the factory to create an object of a certain abstract type and to return the abstract pointer to it, keeping the client from knowing anything about the actual creation of the object.

The fact that the factory returns an abstract pointer to the created object means that the client doesn't have knowledge of the object's type. This implies that there is no need for including any class declarations relating to the concrete type, the client deals at all times with the abstract type. The objects of the concrete type, created by the factory, are accessed by the client only through the abstract interface.

The second implication of this way of creating objects is that when adding new concrete types is needed, all we have to do is modify the client code and make it use a different factory, which is far easier than instantiating a new type, which requires changing the code wherever a new object is created.

The classic implementation for the Abstract Factory pattern is the following:

```
abstract class AbstractProductA
{
	public abstract void operationA1();
	public abstract void operationA2();
}
class ProductA1 extends AbstractProductA
{
	ProductA1(String arg)
	{
		System.out.println("Hello "+arg);
	}
	// Implement the code here
	public void operationA1() { };
	public void operationA2() { };
}
class ProductA2 extends AbstractProductA
{
	ProductA2(String arg)
	{
		System.out.println("Hello "+arg);
	}
	// Implement the code here
	public void operationA1() { };
	public void operationA2() { };
}
abstract class AbstractProductB
{
	public abstract void operationB1();
	public abstract void operationB2();
}
class ProductB1 extends AbstractProductB
{
	ProductB1(String arg)
	{
		System.out.println("Hello "+arg);
	}
	// Implement the code here
}
```

```
class ProductB2 extends AbstractProductB
{
	ProductB2(String arg)
	{
		System.out.println("Hello "+arg);
	}
	// Implement the code here
}
abstract class AbstractFactory
{
	abstract AbstractProductA createProductA();
	abstract AbstractProductB createProductB();
}
class ConcreteFactory1 extends AbstractFactory
{
	AbstractProductA createProductA()
	{
		return new ProductA1("ProductA1");
	}
	AbstractProductB createProductB()
	{
		return new ProductB1("ProductB1");
	}
}
class ConcreteFactory2 extends AbstractFactory
{
	AbstractProductA createProductA()
	{
		return new ProductA2("ProductA2");
	}
	AbstractProductB createProductB()
	{
		return new ProductB2("ProductB2");
	}
}
//Factory creator - an indirect way of instantiating the factories
class FactoryMaker
{
	private static AbstractFactory pf=null;
	static AbstractFactory getFactory(String choice)
	{
		if(choice.equals("a"))
		{
			pf=new ConcreteFactory1();
		}
		else if(choice.equals("b"))
		{
			pf=new ConcreteFactory2();
		}
		return pf;
	}
```

```
}
// Client
public class Client
{
        public static void main(String args[])
        {
                AbstractFactory pf=FactoryMaker.getFactory("a");
                AbstractProductA product=pf.createProductA();
                //more function calls on product
        }
}
```

Issues in implementation:

Factories as Singletons:
An application generally needs only one instance of the ConcreteFactory class per family product. It is best to implement that class as a Singleton.

Creating the products:
The AbstractFactory class only declares the interface for creating the products. It is the task of the ConcreteProduct class to actually create the products. For each family the best idea is applying the Factory Method design pattern. A concrete factory will specify its products by overriding the factory method for each of them. Even if the implementation might seem simple, using this idea will mean defining a new concrete factory subclass for each product family, even if the classes are similar in most aspects.

For simplifying the code and increase the performance the Prototype design pattern can be used instead of Factory Method, especially when there are many product families. In this case the concrete factory is initiated with a prototypical instance of each product in the family and when a new one is needed instead of creating it, the existing prototype is cloned. This approach eliminates the need for a new concrete factory for each new family of products.

Extending the factories:
The operation of changing a factory in order for it to support the creation of new products is not easy. What can be done to solve this problem is, instead of a CreateProduct method for each product, to use a single Create method that takes a parameter that identifies the type of product needed. This approach is more flexible, but less secure. The problem is that all the objects returned by the Create method will have the same interface, that is the one corresponding to the type returned by the Create method and the client will not always be able to correctly detect to which class the instance actually belongs.

Sample Code:
Let's take the UI toolkit concept on to our Java code example. We'll create a client application that needs to create a window.

First, we'll need to create our Window interface. Window is our AbstractProduct.

```
//Our AbstractProduct
public interface Window
{
        public void setTitle(String text);
        public void repaint();
```

```
}
```

Let's create two implementations of the Window, as our ConcreteProducts. One for Microsoft Windows:

```
//ConcreteProductA1
public class MSWindow implements Window
{
	public void setTitle()
	{
		//MS Windows specific behaviour
	}
	public void repaint()
	{
		//MS Windows specific behaviour
	}
}
```

And one for Mac OSX:

```
//ConcreteProductA2
public class MacOSXWindow implements Window
{
	public void setTitle()
	{
		//Mac OSX specific behaviour
	}
	public void repaint()
	{
		//Mac OSX specific behaviour
	}
}
```

Now we need to provide our factories. First we'll define our AbstractFactory. For this example, let's say they just create Windows:

```
//AbstractFactory
public interface AbstractWidgetFactory
{
	public Window createWindow();
}
```

Next we need to provide ConcreteFactory implementations of these factories for our two operating systems. First for MS Windows:

```
//ConcreteFactory1
public class MsWindowsWidgetFactory
{
	//create an MSWindow
	public Window createWindow()
	{
```

```
                MSWindow window = new MSWindow();
                return window;
        }
}
```

And for MacOSX:

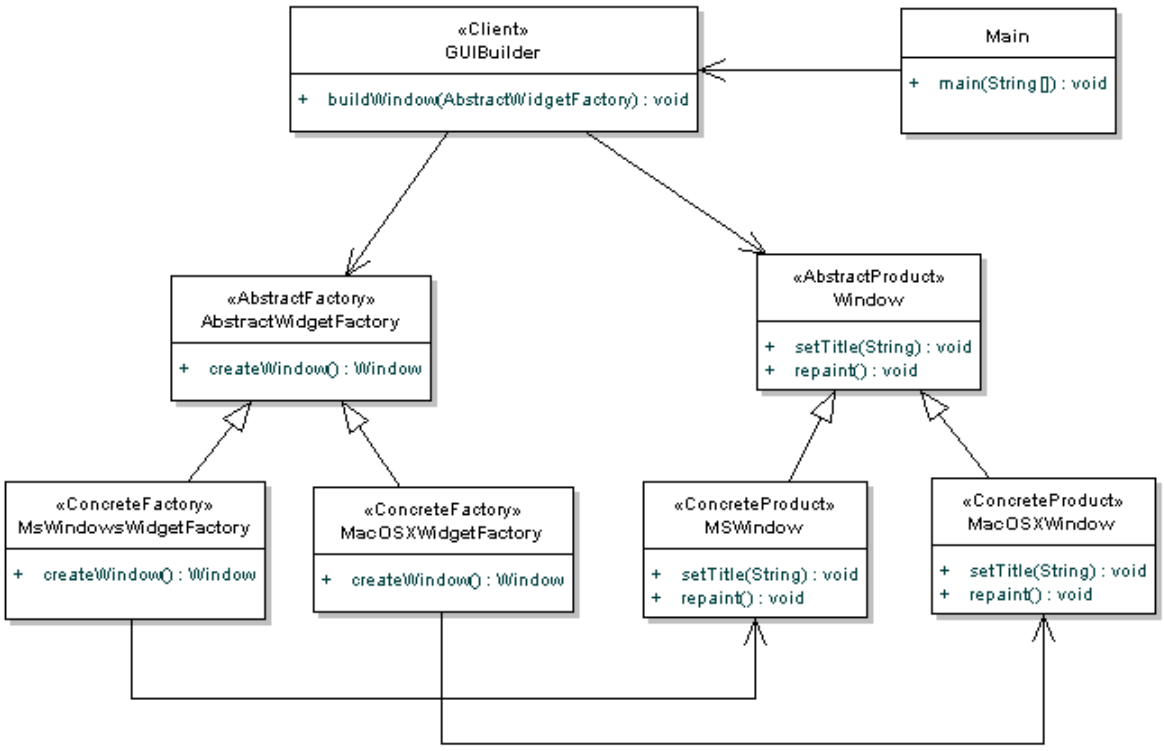
```
//ConcreteFactory2
public class MacOSXWidgetFactory
{
        //create a MacOSXWindow
        public Window createWindow()
        {
                MacOSXWindow window = new MacOSXWindow();
                return window;
        }
}
```

Finally we need a client to take advantage of all this functionality.

```
//Client
public class GUIBuilder
{
        public void buildWindow(AbstractWidgetFactory widgetFactory)
        {
                Window window = widgetFactory.createWindow();
                window.setTitle("New Window");
        }
}
```

We need some way to specify which type of AbstractWidgetFactory to our GUIBuilder. This is usually done with a switch statement similar to the code below:

```
public class Main
{
        public static void main(String[] args)
        {
                GUIBuilder builder = new GUIBuilder();
                AbstractWidgetFactory widgetFactory = null;
                //check what platform we're on
                if(Platform.currentPlatform()=="MACOSX")
                {
                        widgetFactory  = new MacOSXWidgetFactory();
                }
                else
                {
                        widgetFactory  = new MsWindowsWidgetFactory();
                }
                builder.buildWindow(widgetFactory);
        }
}
```



Known Uses:

1) InterViews uses the "kit" suffix to denote AbstractFactory classes. It defines WidgetKit and DialogKit abstract factories to generate user interface objects.
2) ET++ uses the Abstract Factory pattern to achieve portability across different window systems.

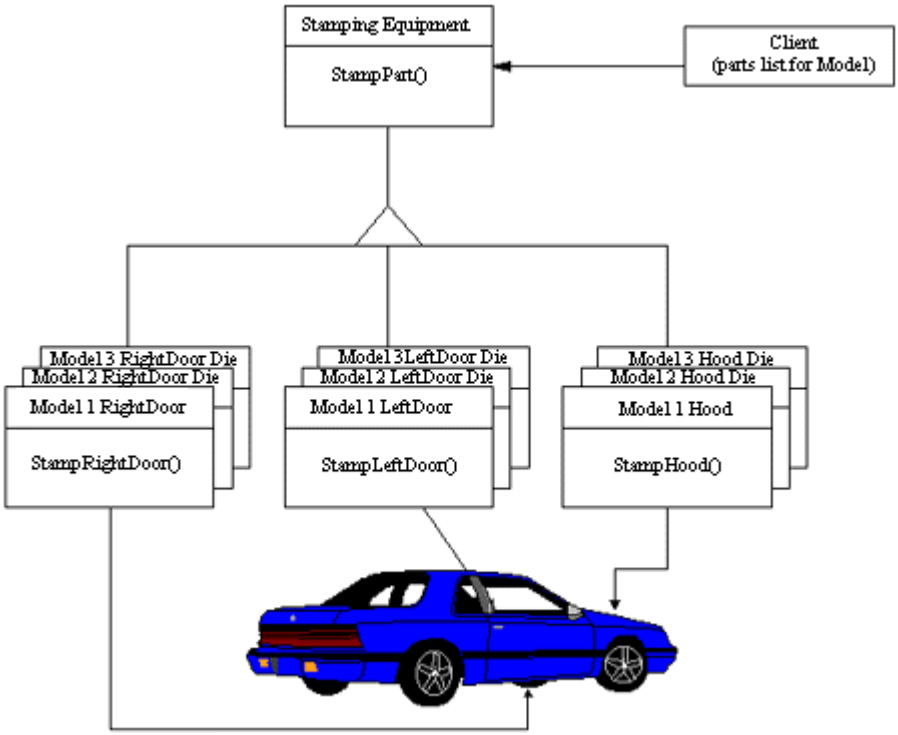
Related Patterns:

AbstractFactory classes are often implemented with factory methods but they can also be implemented using Prototype. A concrete factory is often a Singleton.

Example Abstract Factory classes in java API:
javax.xml.parsers.DocumentBuilderFactory
javax.xml.transform.TransformerFactory
javax.xml.xpath.XPathFactory

Non-software 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.


Stamping Equipment
StampPart()
Client
(parts list for Model)
Model 3 RightDoor Die
Model 2 RightDoor Die
Model 1 RightDoor
StampRightDoor()
Model 3 LeftDoor Die
Model 2 LeftDoor Die
Model 1 LeftDoor
StampLeftDoor()
Model 3 Hood Die
Model 2 Hood Die
Model 1 Hood
StampHood()

Factory Method Pattern

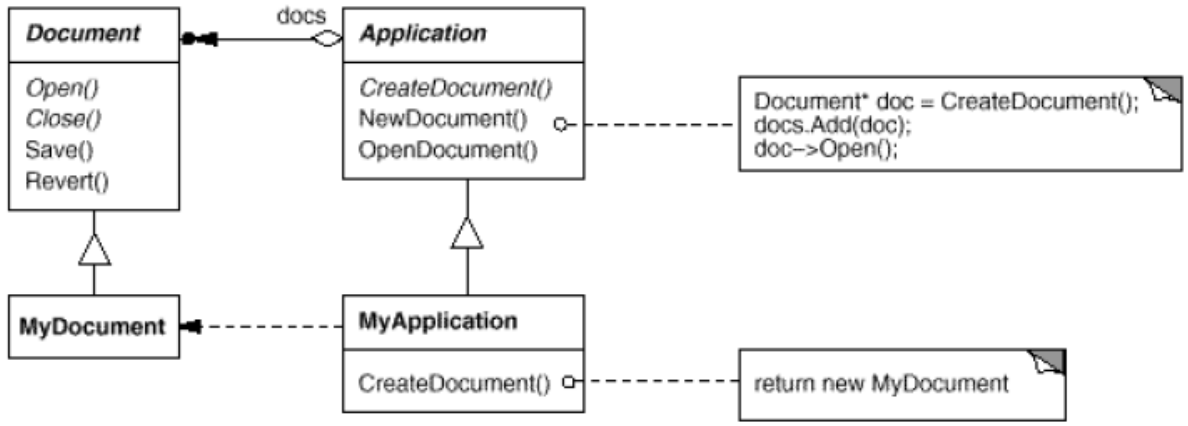
Intent:
Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As:
Virtual Constructor

Motivation:
Take into consideration a framework for desktop applications. Such applications are meant to work with documents. A framework for desktop applications contains definitions for operations such as opening, creating and saving a document. The basic classes are abstract ones, named Application and Document, their clients having to create subclasses from them in order to define their own applications. For generating a drawing application, for example, they need to define the DrawingApplication and DrawingDocument classes. The Application class has the task of managing the documents, taking action at the request of the client (for example, when the user selects the open or save command form the menu).

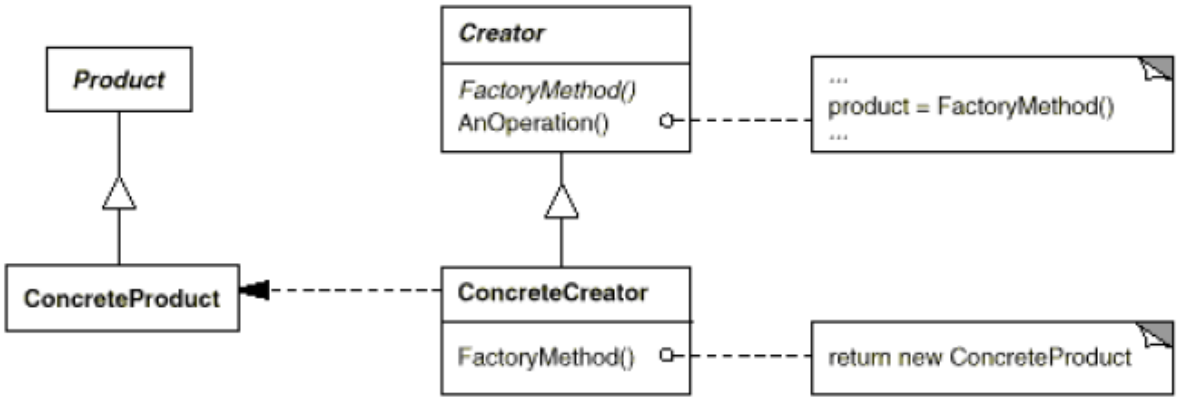
Because the Document class that needs to be instantiated is specific to the application, the Application class does not know it in advance, so it doesn't know what to instantiate, but it does know when to instantiate it. The framework needs to instantiate a certain class, but it only knows abstract classes that can't be instantiated.

The Factory Method design pattern solves the problem by putting all the information related to the class that needs to be instantiated into an object and using them outside the framework, as you can see below:



Applicability:
1. A class can’t anticipate the class of objects it must create.
2. A class wants its sub classes to specify the objects it creates.
3. Classes delegate responsibility to one of several helper subclasses.

Structure:



Participants:
The classes that take part in Factory Method pattern are:

Product: Defines the interface of objects the factory method creates.
ConcreteProduct: Implements the *Product* interface.
Creator: Declares the factory method, which returns an object of the type *Product*. Creator may also provide a default implementation of the factory method which returns a default *ConcreteProduct* object.
ConcreteCreator: Overrides the factory method to return an instance of a *ConcreteProduct*.

Collaborations:
Creator relies on its subclasses to provide an implementation for the factory method so that it returns an instance of the appropriate *ConcreteProduct* class.

Consequences:
The benefits and drawbacks of factory method pattern are as follows:

- The main reason for which the factory pattern is used is that it introduces a separation between the application and a family of classes (it introduces weak coupling instead of tight coupling hiding concrete classes from the application). It provides a simple way of extending the family of products with minor changes in application code.
- It provides customization hooks. When the objects are created directly inside the class it's hard to replace them by objects which extend their functionality. If a factory is used instead to create a family of objects the customized objects can easily replace the original objects, configuring the factory to create them.
- The factory has to be used for a family of objects. If the classes doesn't extend common base class or interface they can not be used in a factory design template.

Implementation:
All concrete products are subclasses of the Product class, so all of them have the same basic implementation, at some extent. The Creator class specifies all standard and generic behavior of the products and when a new product is needed, it sends the creation details that are supplied by the client to the ConcreteCreator. Having this diagram in mind, it is easy for us now to produce the code related to it. Here is how the implementation of the classic Factory method should look:

```
public interface Product { }
```

```
public class ConcreteProduct implements Product {  }

public abstract class Creator
{
	public void anOperation()
	{
		Product product = factoryMethod();
	}
	protected abstract Product factoryMethod();
}

public class ConcreteCreator extends Creator
{
	protected Product factoryMethod()
	{
		return new ConcreteProduct();
	}
}

public class Client
{
	public static void main( String arg[] )
	{
		Creator creator = new ConcreteCreator();
		creator.anOperation();
	}
}
```

Sample Code:
Let's consider a Vehicle interface and 2 of its implementations namely Car and Vehicle:

```
interface Vehicle
{
	public void drive();
	public void clean();
}

class Car implements Vehicle
{
	@Override
	public void drive()
	{
		System.out.println("Driving a car...");
	}
	@Override
	public void clean()
	{
		System.out.println("Cleaning a car...");
	}
}
```

```
class Bus implements Vehicle
{
	@Override
	public void drive()
	{
		System.out.println("Driving a Bus...");
	}
	@Override
	public void clean()
	{
		System.out.println("Cleaning a Bus...");
	}
}
```

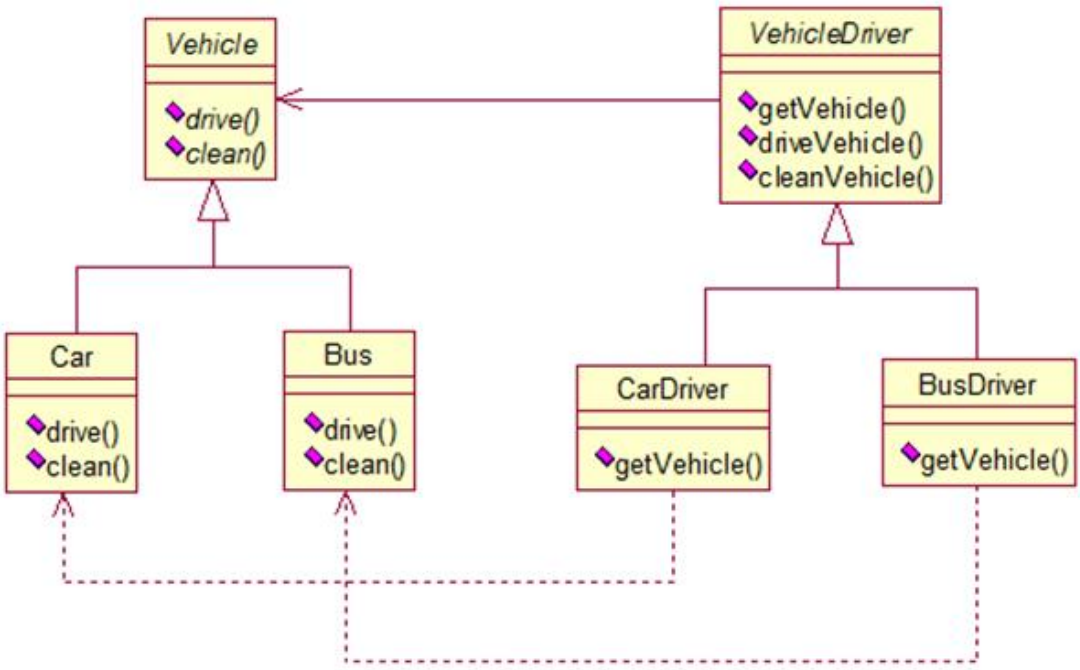
And to drive() and clean() the Vehicle we would use a VehicleDriver.

Let's consider the implementation and of VehicleDriver:

```
abstract class VehicleDriver
{
	public abstract Vehicle getVehicle();
	public void driveVehicle()
	{
		getVehicle().drive();
	}
	public void cleanVehicle()
	{
		getVehicle().clean();
	}
}

class CarDriver extends VehicleDriver
{
	@Override
	public Vehicle getVehicle()
	{
		return new Car();
	}
}

class BusDriver extends VehicleDriver
{
	@Override
	public Vehicle getVehicle()
	{
		return new Bus();
	}
}
```

In the above VehicleDriver implementation the getVehicle() method is the factory method which is overridden by the CarDriver and Busdriver to return Car and Businstances respectively. In this way

the programmer would be more concerned about using the VehicleDriver abstraction and need not be concerned about its different implementations.



Known Uses:

1. Class *View* in the Smalltalk-80 Model/View/Controller framework has a method defaultController that creates a controller, and this might appear to be a factory method. But subclasses of View specify the class of their default controller by defining defaultControllerClass, which returns the class from which defaultController creates instances. So defaultControllerClass is the real factory method, that is, the method that subclasses should override.
2. In Smalltalk-80 the factory method parserClass defined by Behavior (a superclass of all objects representing classes). This enables a class to use a customized parser for its source code.
3. The Orbix ORB system from IONA Technologies uses Factory Method to generate an appropriate type of proxy when an object requests a reference to a remote object.

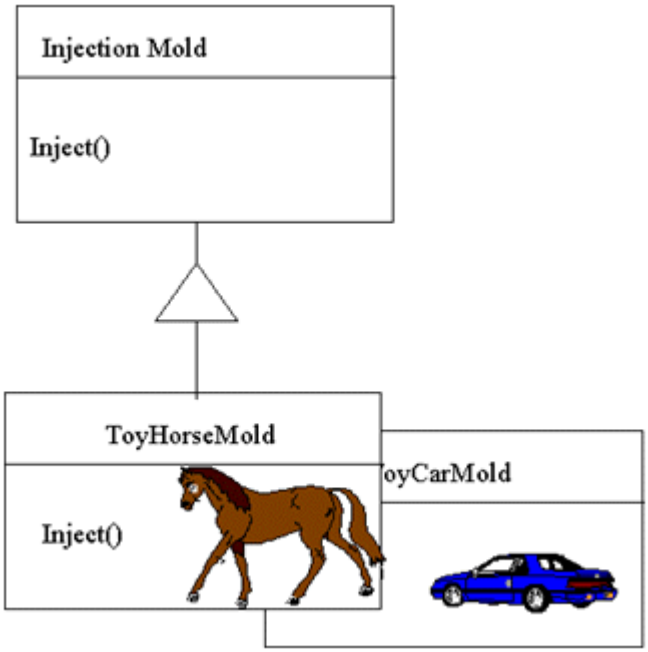
Related Patterns:

- Abstract Factory is often implemented with factory methods.
- Factory methods are usually called within Template Methods. In the Document example above, NewDocument() is a template method.
- Prototypes don't require subclassing Creator. Instead, they often require an Initialize operation on the product class. Factory Method doesn't require such operation.

Example Factory Method classes in java API:
java.util.Calendar
java.util.ResourceBundle
java.text.NumberFormat

java.nio.charset.Charset
java.net.URLStreamHandlerFactory

Non-software example:
The *Factory Method* defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Builder Pattern

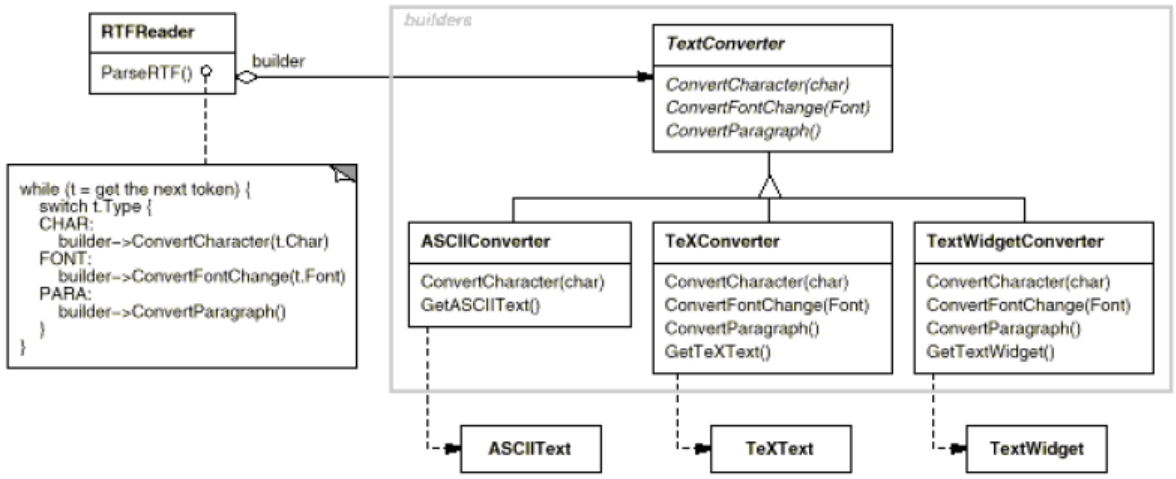
Intent:
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation:
A RTF (Rich Text Format) reader application should be able to convert a RTF document to several formats like: plain text or TeX representation or into a text widget which allows users to directly interact. The problem here is, the number of possible conversions is open ended. So, it should be easy to add a new conversion without modifying the reader.

A solution to this problem is, the RTFReader class uses a TextConverter object that converts RTF to other textual repesentations. Whenever the RTReader encounters a RTF token, it notifies the TextConverter object about it. TextConverter object is responsible both for converting the data and for representing the token in a specific format.

The subclasses of TextConverter are specialized for different conversions. For example, the ASCIIConverter converts RTF to plain text. TeXConverter will produce a TeX representation and TextWidgetConverter will produce a complex user interface object that lets the user see and edit the text.

Each converter class takes the responsibility for creating and assembling a complex object and puts it behind an abstract interface. Each converter class is called a **builder** in the pattern, and the reader is called the **director**. The Builder pattern separates the algorithm for interpreting a textual format (reader for RTF documents) from how a converted format gets created and represented.



Applicability:
Use the Builder pattern when:

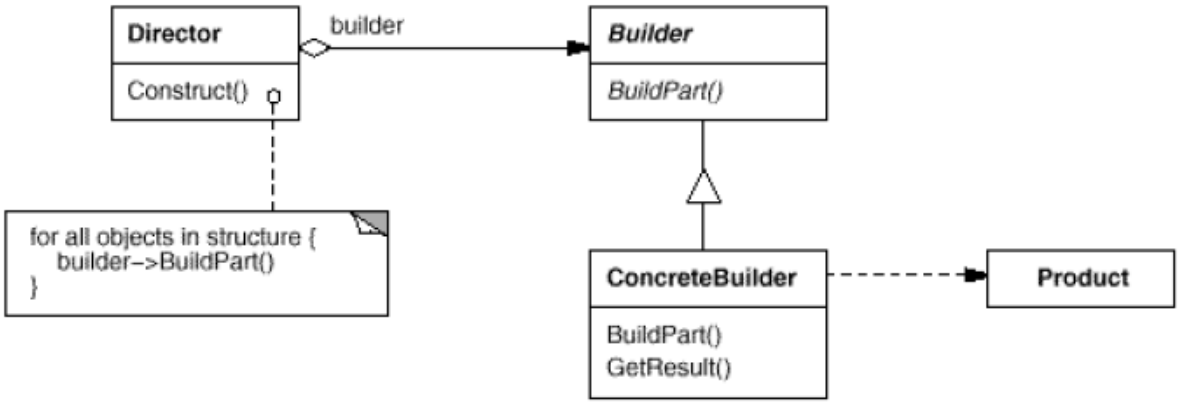
- The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
- The construction process must allow different representations for the object that is constructed.

Examples where Builder pattern can be applied:

- Vehicle manufacturer

- Student exams

Structure:



Participants:
The particpants in the Builder pattern are:

Builder: Provides an abstract interface for creating parts of a Product object.
ConcreteBuilder: Constructs and assembles the parts of the Product by implementing the Builder interface. Defines and keeps track of the representation it creates. Provides an interface for retrieving the product.
Director: Constructs an object using the Builder interface.
Product: Represents the complex object under construction. Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

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 the requests from the Director and adds parts to the product.
- The client retrieves the product from the Builder.

Consequences:
The benefits and pitfalls of Builder pattern are:
- It let's you vary the product's internal representation.
- It isolates code for construction and representation.
- It gives you finer control over the construction process.

Implementation:
The client, that may be either another object or the actual client that calls the main() method of the application, initiates the Builder and Director class. The Builder represents the complex object that needs to be built in terms of simpler objects and types. The constructor in the Director class receives a Builder object as a parameter from the Client and is responsible for calling the appropriate methods of the Builder class. In order to provide the Client with an interface for all concrete Builders, the Builder class should be an abstract one. This way you can add new types of complex objects by only defining the structure and reusing the logic for the actual construction process. The Client is the only one that needs to know about the new types, the Director needing to know which methods of the Builder to call.

The Client needs to convert a document from RTF format to ASCII format. There for, it calls the method createASCIIText that takes as a parameter the document that will be converted. This method calls the concrete builder, ASCIIConverter, that extends the Builder, TextConverter, and overrides its two methods for converting characters and paragraphs, and also the Director, RTFReader, that parses the document and calls the builder's methods depending on the type of token encountered. The product, the ASCIIText, is built step by step, by appending converted characters. The implementation code is as follows:

```
//Abstract Builder
class abstract class TextConverter
{
        abstract void convertCharacter(char c);
        abstract void convertParagraph();
}

// Product
class ASCIIText
{
        public void append(char c)
        { //Implement the code here }
}

//Concrete Builder
class ASCIIConverter extends TextConverter
{
        ASCIIText asciiTextObj;   //resulting product
        /*converts a character to target representation and appends to the resulting object*/
        void convertCharacter(char c)
        {
                char asciiChar = new Character(c).charValue();
                //gets the ascii character
                asciiTextObj.append(asciiChar);
        }
        void convertParagraph() {}
        ASCIIText getResult()
        {
                return asciiTextObj;
        }
}

//This class abstracts the document object
class Document
{
        static int value;
        char token;
        public char getNextToken()
        {
                //Get the next token
                return token;
        }
}
```

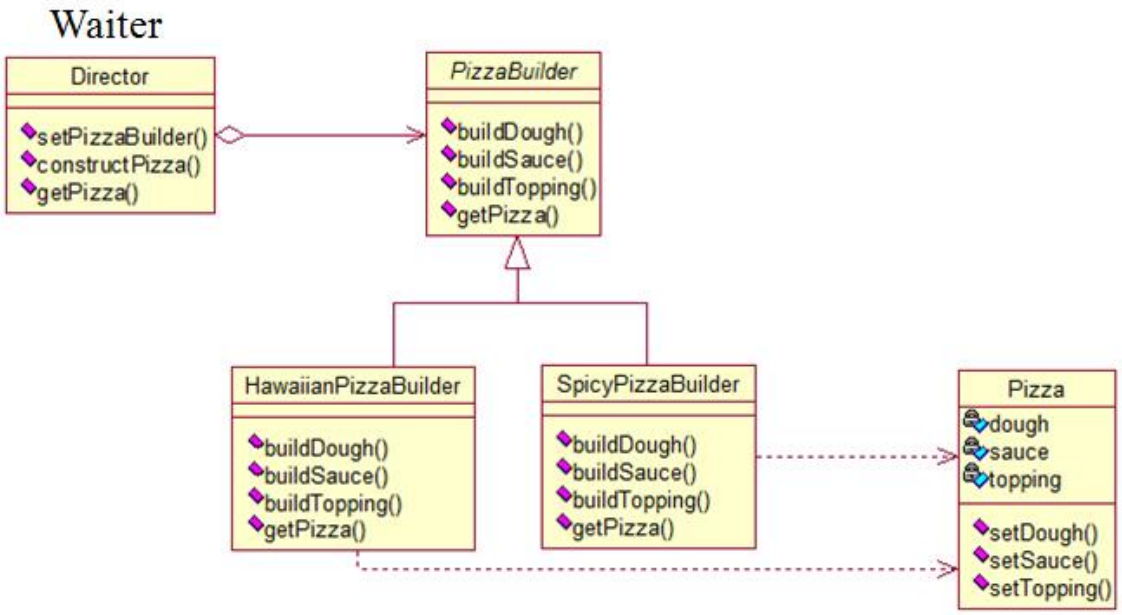
```
//Director
class RTFReader
{
        private static final char EOF='0'; //Delimitor for End of File
        final char CHAR='c';
        final char PARA='p';
        char t;
        TextConverter builder;
        RTFReader(TextConverter obj)
        {
                builder=obj;
        }
        void parseRTF(Document doc)
        {
                while ((t=doc.getNextToken())!= EOF)
                {
                        switch (t)
                        {
                                case CHAR: builder.convertCharacter(t);
                                case PARA: builder.convertParagraph();
                        }
                }
        }
}

//Client
public class Client
{
        void createASCIIText(Document doc)
        {
                ASCIIConverter asciiBuilder = new ASCIIConverter();
                RTFReader rtfReader = new RTFReader(asciiBuilder);
                rtfReader.parseRTF(doc);
                ASCIIText asciiText = asciiBuilder.getResult();
        }
        public static void main(String args[])
        {
                Client client=new Client();
                Document doc=new Document();
                client.createASCIIText(doc);
                system.out.println("This is an example of Builder Pattern");
        }
}
```

Implementation issues:

Abstract class for products:
In practice the products created by the concrete builders have a structure significantly different, so there is no reason to derive different products from common parent class. This also distinguishes the Builder pattern from the Abstract Factory pattern which creates objects derived from a common type.

Assembly and construction interface:
Builders construct their products in step-by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. A key design issue concerns the model for the construction and assembly process. A model where the results of construction requests are simply appended to the product is usually sufficient. In the RTF example, the builder converts and appends the next token to the text it has converted so far.

Sample Code:




```
/* "Product" */
class Pizza
{
        private String dough = "";
        private String sauce = "";
        private String topping = "";
        public void setDough(String dough)     { this.dough = dough; }
        public void setSauce(String sauce)     { this.sauce = sauce; }
        public void setTopping(String topping) { this.topping = topping; }
}

/* "Builder" */
abstract class PizzaBuilder
{
        protected Pizza pizza;
        public Pizza getPizza() { return pizza; }
        public void createNewPizzaProduct() { pizza = new Pizza(); }
        public abstract void buildDough();
        public abstract void buildSauce();
        public abstract void buildTopping();
}

/* "ConcreteBuilder" */
```

```
class HawaiianPizzaBuilder extends PizzaBuilder
{
        public void buildDough()   { pizza.setDough("cross"); }
        public void buildSauce()   { pizza.setSauce("mild"); }
        public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder
{
        public void buildDough()   { pizza.setDough("pan baked"); }
        public void buildSauce()   { pizza.setSauce("hot"); }
        public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}

/* "Director" */
class Waiter
{
        private PizzaBuilder pizzaBuilder;
        public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
        public Pizza getPizza() { return pizzaBuilder.getPizza(); }
        public void constructPizza()
        {
                pizzaBuilder.createNewPizzaProduct();
                pizzaBuilder.buildDough();
                pizzaBuilder.buildSauce();
                pizzaBuilder.buildTopping();
        }
}

/* A customer ordering a pizza. */
class Client
{
        public static void main(String[] args)
        {
                Waiter waiter = new Waiter();
                PizzaBuilder hawaiian_pizzabuilder = new HawaiianPizzaBuilder();
                PizzaBuilder spicy_pizzabuilder = new SpicyPizzaBuilder();
                waiter.setPizzaBuilder( hawaiian_pizzabuilder );
                waiter.constructPizza();
                Pizza pizza = waiter.getPizza();
        }
}
```

Known Uses:

Builder pattern is used commonly across Smalltalk-80:

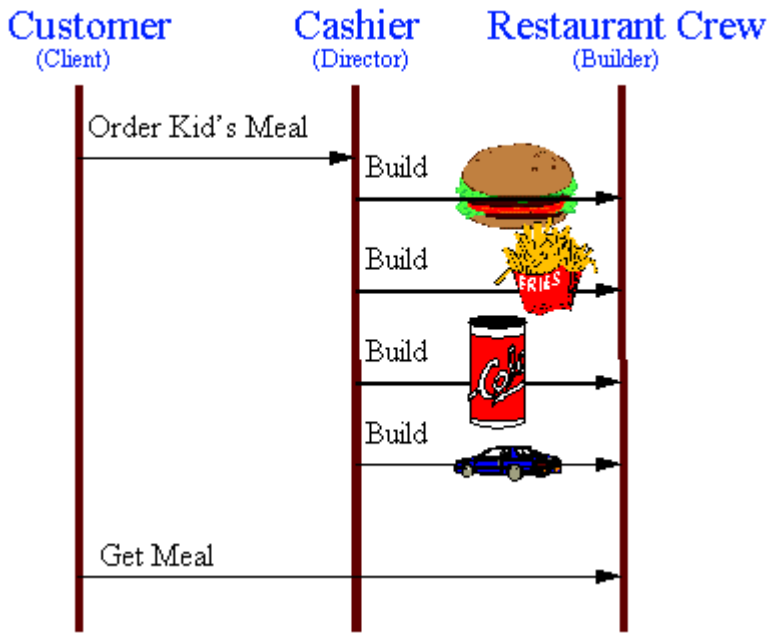
- The Parser class in the compiler subsystem is a Director that takes a ProgramNodeBuilder object as an argument.
- ClassBuilder is a builder that classes use to create subclassses for themselves. In this case a Class is both Director and a Builder.
- ByteCodeStream is a builder that creates a compiled method as a byte array.

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 a builder often builds.

Example Builder classes in java API:
java.lang.StringBuilder (unsynchronized)
java.lang.StringBuffer (synchronized)
java.nio.ByteBuffer (also CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
javax.swing.GroupLayout.Group
All implementations of java.lang.Appendable

Non-software example:
The *Builder* pattern separates the construction of a complex object from its representation, so that the same construction process can create different representation. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, coke, and toy car). Note that there can be variation in the contents of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.



Prototype Pattern

Intent:
To specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Motivation:
A prototype is a template of any object before the actual object is constructed. In java also, it holds the same meaning. Prototype design pattern is used in scenarios where application needs to create a number of instances of a class, which has almost same state or differs very little.

In this design pattern, an instance of actual object (i.e. prototype) is created on starting, and thereafter whenever a new instance is required, this prototype is cloned to have another instance. The main advantage of this pattern is to have minimal instance creation process which is much costly than cloning process.

Some of the situations where the prototype pattern can be applied are given below:

Example 1:

In building stages for a game that uses a maze and different visual objects that the character encounters it is needed a quick method of generating the haze map using the same objects: wall, door, passage, room... The Prototype pattern is useful in this case because instead of hard coding (using new operation) the room, door, passage and wall objects that get instantiated, CreateMaze method will be parameterized by various prototypical room, door, wall and passage objects, so the composition of the map can be easily changed by replacing the prototypical objects with different ones.

The Client is the CreateMaze method and the ConcretePrototype classes will be the ones creating copies for different objects.

Example 2:

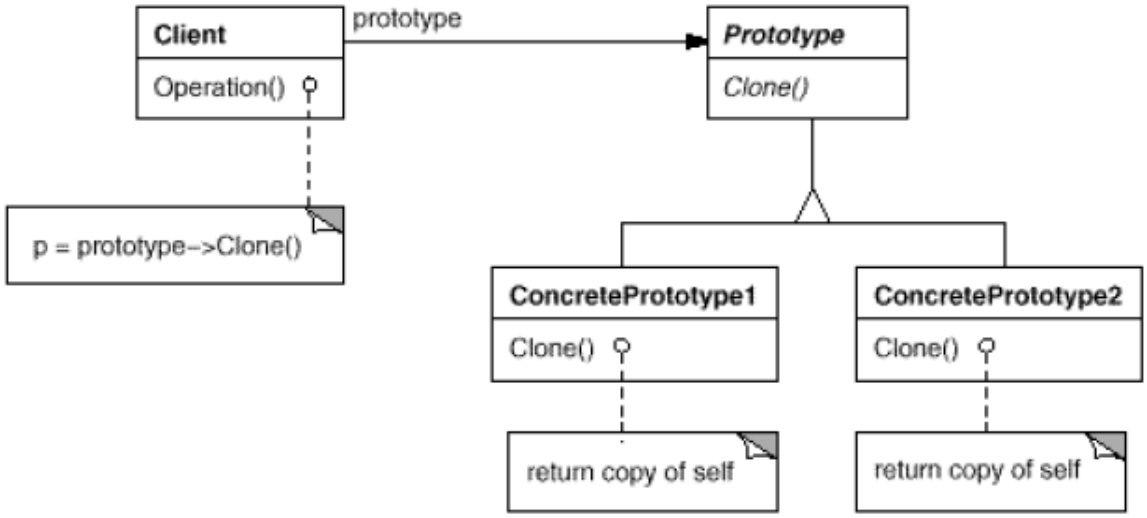
Suppose we are doing a sales analysis on a set of data from a database. Normally, we would copy the information from the database, encapsulate it into an object and do the analysis. But if another analysis is needed on the same set of data, reading the database again and creating a new object is not the best idea. If we are using the Prototype pattern then the object used in the first analysis will be cloned and used for the other analysis.

The Client methods process the object that encapsulates information from the database. The ConcretePrototype class will be class that creates the object after extracting data from the database, will copy it into objects used for analysis.

Applicability:
Use prototype pattern when a system should be independent of how its products are created, composed and represented and:

- When the class to be instantiated is specified at runtime or
- To avoid building a hierarchy of factory classes or
- It is more convenient to clone an object rather than creating a new object.

Structure:



Participants:
The classes participating to the Prototype Pattern are:

Client - creates a new object by asking a prototype to clone itself.
Prototype - declares an interface for cloning itself.
ConcretePrototype - implements the operation for cloning itself.

Collaborations:
A client asks a prototype to clone itself.

Consequences:
Benefits of prototype pattern are:

- Adding and removing products at runtime.
- Specifying new objects by varying values.
- Specifying new objects by varying structure.
- Reduced subclassing.
- Configuring an application with classes dynamically.

Main drawback of prototype pattern is that each sub class of Prototype must implement the clone operation, which may be difficult.

Implementation:

```
abstract class AbstractProduct implements Cloneable
{
        public static AbstractProduct thePrototype;
        public static AbstractProduct makeProduct()
        {
                try
                {
                        return (AbstractProduct) thePrototype.clone();
                 }
                catch(CloneNotSupportedException e)
```

```
                {
                        return null;
                 }
        }
}

class ConcreteProductA extends AbstractProduct { }

class ConcreteProductB extends AbstractProduct { }

public class PrototypeDemo
{
        public static void main(String[] args)
        {
                AbstractProduct.thePrototype = new ConcreteProductA();
                AbstractProduct product = AbstractProduct.makeProduct();
                System.out.println(product);
        }
}
```

Implementation Issues:

Using a prototype manager:

When the application uses a lot of prototypes that can be created and destroyed dynamically, a registry of available prototypes should be kept. This registry is called the prototype manager and it should implement operations for managing registered prototypes like registering a prototype under a certain key, searching for a prototype with a given key, removing one from the register, etc. The clients will use the interface of the prototype manager to handle prototypes at run-time and will ask for permission before using the Clone() method.
There is not much difference between an implementation of a prototype which uses a prototype manager and a factory method implemented using class registration mechanism. Maybe the only difference consists in the performance.

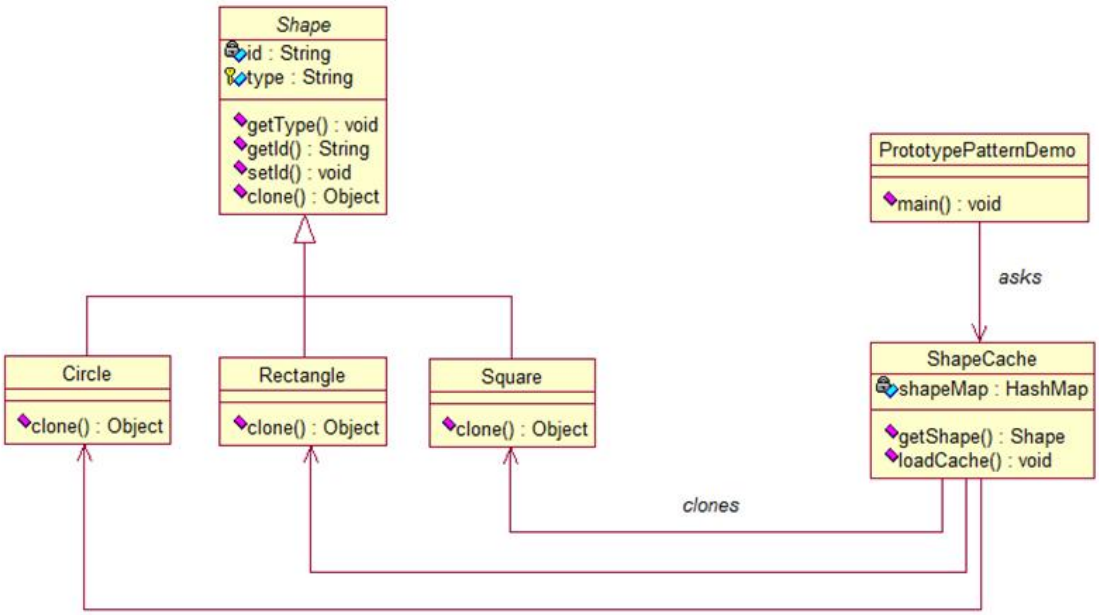
Implementing the Clone operation:

A small discussion appears when talking about how deep or shallow a clone should be: a deep clone clones the instance variables in the cloning object while a shallow clone shares the instance variables between the clone and the original. Usually, a shallow clone is enough and very simple, but cloning complex prototypes should use deep clones so the clone and the original are independent, a deep clone needing its components to be the clones of the complex object's components.

Initializing clones:

There are cases when the internal states of a clone should be initialized after it is created. This happens because these values cannot be passed to the Clone() method, that uses an interface which would be destroyed if such parameters were used. In this case the initialization should be done by using setting and resetting operations of the prototype class or by using an initializing method that takes as parameters the values at which the clone's internal states should be set.

Sample Code:
We're going to create an abstract class *Shape* and concrete classes extending the *Shape* class. A class *ShapeCache* is defined as a next step which stores shape objects in a *Hashtable* and returns their clone when requested.



Create an abstract class implementing *Clonable* interface.

Shape.java

```
public abstract class Shape implements Cloneable
{
	private String id;
	protected String type;
	abstract void draw();
	public String getType()
	{
		return type;
	}
	public String getId()
	{
	 return id;
	}
	public void setId(String id)
	{
		this.id = id;
	}
	public Object clone()
	{
		Object clone = null;
		try
		{
			clone = super.clone();
```

```
                }
                catch (CloneNotSupportedException e)
                {
                        e.printStackTrace();
                }
                return clone;
        }
}
```

Create concrete classes extending the above class.

Rectangle.java

```
public class Rectangle extends Shape
{
        public Rectangle()
        {
                type = "Rectangle";
        }
        @Override
        public void draw()
        {
                System.out.println("Inside Rectangle::draw() method.");
        }
}
```

Square.java

```
public class Square extends Shape
{
        public Square()
        {
                type = "Square";
        }
        @Override
        public void draw()
        {
                System.out.println("Inside Square::draw() method.");
        }
}
```

Circle.java

```
public class Circle extends Shape
{
        public Circle()
        {
                type = "Circle";
        }
        @Override
        public void draw()
```

```
	{
		System.out.println("Inside Circle::draw() method.");
	}
}
```

Create a class to get concreate classes from database and store them in a *Hashtable*.

ShapeCache.java

```
import java.util.Hashtable;
public class ShapeCache
{
	private static Hashtable<String, Shape> shapeMap = new Hashtable<String, Shape>();
	public static Shape getShape(String shapeId)
	{
		Shape cachedShape = shapeMap.get(shapeId);
		return (Shape) cachedShape.clone();
	}
	public static void loadCache()
	{
		Circle circle = new Circle();
		circle.setId("1");
		shapeMap.put(circle.getId(),circle);
		Square square = new Square();
		square.setId("2");
		shapeMap.put(square.getId(),square);
		Rectangle rectangle = new Rectangle();
		rectangle.setId("3");
		shapeMap.put(rectangle.getId(),rectangle);
	}
}
```

PrototypePatternDemo uses *ShapeCache* class to get clones of shapes stored in a *Hashtable*.

PrototypePatternDemo.java

```
public class PrototypePatternDemo
{
	public static void main(String[] args)
	{
		ShapeCache.loadCache();
		Shape clonedShape = (Shape) ShapeCache.getShape("1");
		System.out.println("Shape : " + clonedShape.getType());
		Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
		System.out.println("Shape : " + clonedShape2.getType());
		Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
		System.out.println("Shape : " + clonedShape3.getType());
	}
}
```

Known Uses:
The first widely known application of the pattern in an object-oriented language was in ThingLab, where users could form a composite object and then promote it to a prototype by installing it in a library of reusable objects. Prototype is used in Etgdb. Etgdb is a debugger front-end based on ET++ that provides a point-and-click interface to different line-oriented debuggers. The "interaction technique library" in Mode Composer stores prototypes of objects that support various interaction techniques.

Related Patterns:
Prototype and Abstract Factory patterns can be used together. Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.

Non-software example:
The *Prototype* pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive, and does not participate in copying itself. The mitotic division of a cell, resulting in two identical cells, is an example of a prototype that plays an active role in copying itself and thus, demonstrates the *Prototype* pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.