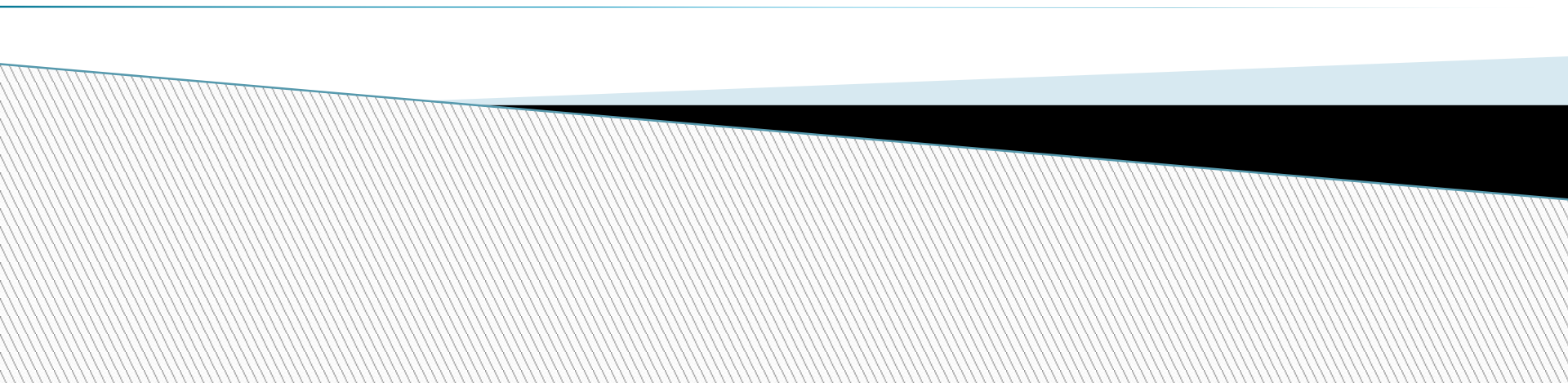


# Design Patterns - Creational



# Creational Patterns

# Creational Patterns

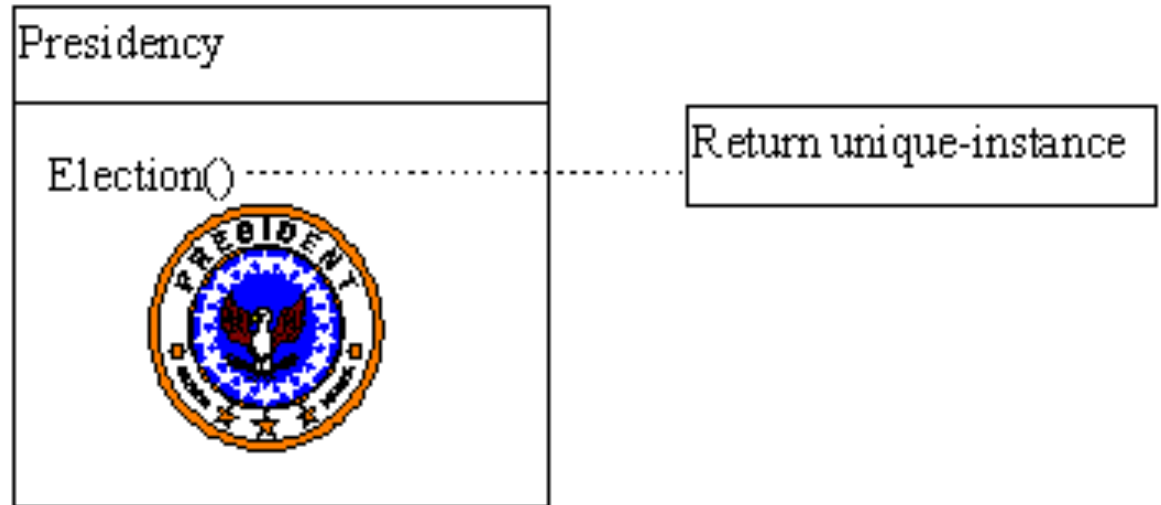
▶ concern process of object creation

- Singleton
- Factory Method
- Prototype
- Abstract Factory
- Builder

# Singleton

# SINGLETON

Object Diagram for  
*Singleton* using  
Presidency 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.

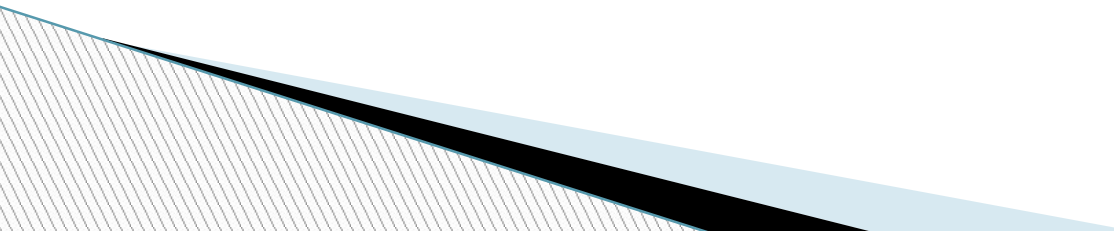
# SINGLETON

## Intent

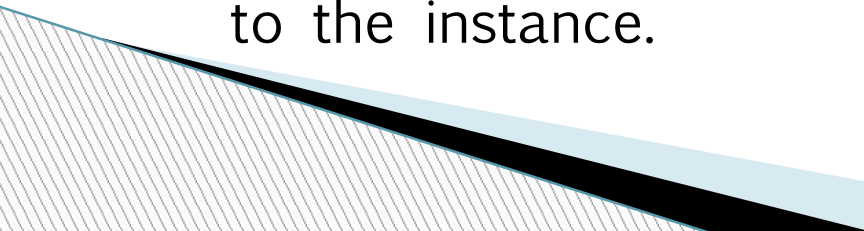
- ▶ Ensure a class has only one instance, and provide a global point of access to it

## Problem

Application needs one, and only one, instance of an object and that the instance is easily accessible. Additionally, lazy initialization and global access are necessary.



## Solution

- ▶ 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 ( by intercepting request to create new objects ), and it provide a way to access the instance.
1. Make the class of the single instance object responsible for creation, initialization, access, and enforcement.
  2. Declare the instance as a private static data member.
  3. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.
- 

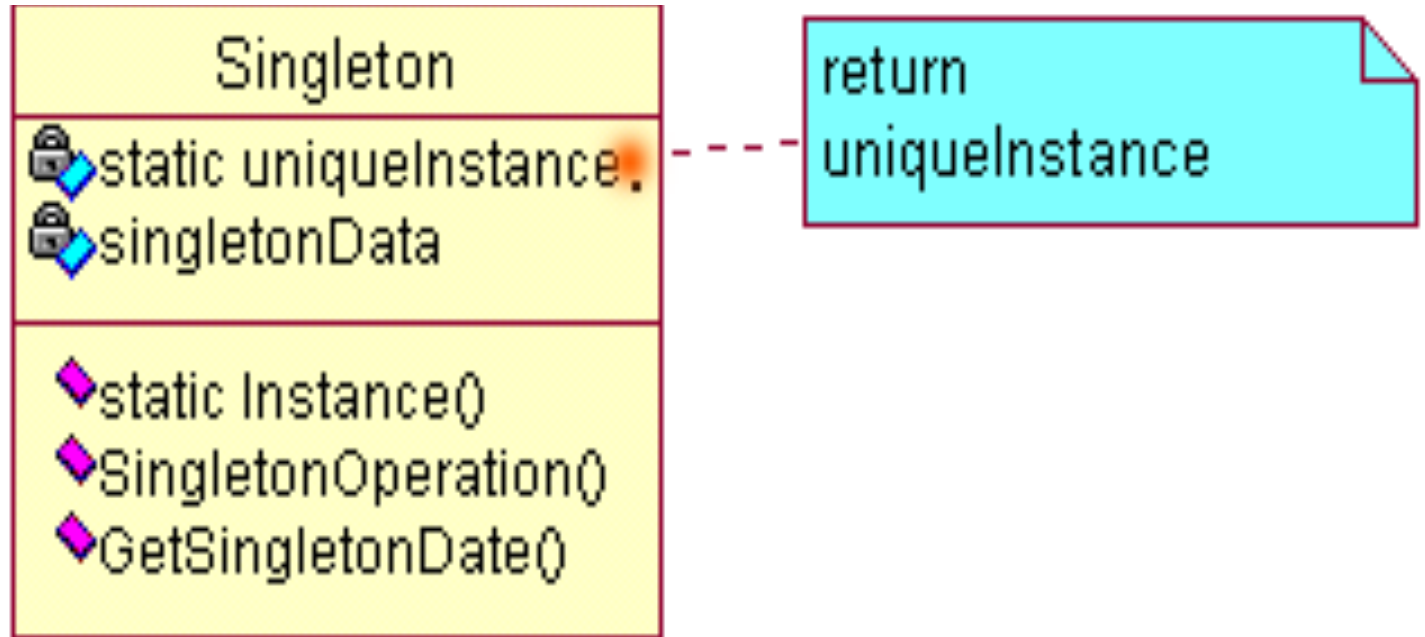
# Applicability

## Use 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 sub-classing, and clients should be able to use and extended instance without modifying their code.



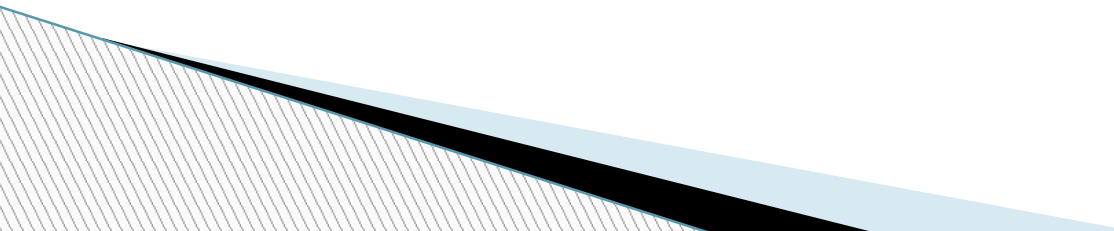
## Structure



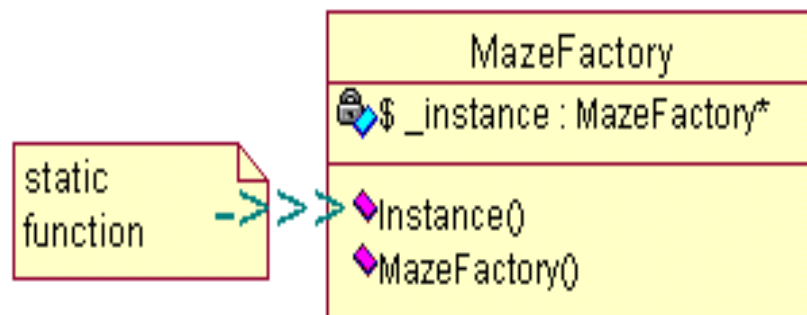
## Collaborations

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

# Consequences

- ▶ Controlled access to sole instance
  - ▶ Reduced name space
  - ▶ Permits refinement of operations and representation
  - ▶ Permits a variable number of instances
  - ▶ More flexible than class operations
- 

# Implementation



```
MazeFactory* MazeFactory::_instance = 0;
```

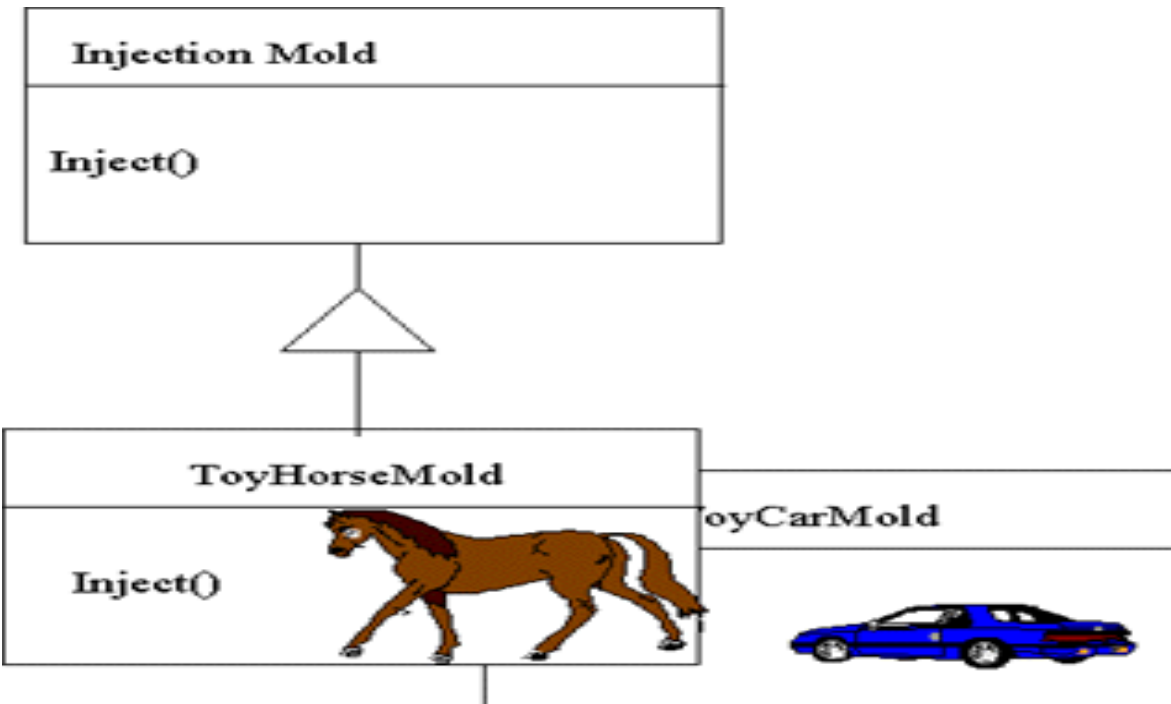
```
MazeFactory* MazeFactory::Instance () {  
    if (_instance == 0) {  
        _instance = new MazeFactory;  
    }  
    return _instance;  
}
```

```
MazeFactory* MazeFactory::Instance () {  
    if (_instance == 0) {  
        const char* mazeStyle = getenv("MAZESTYLE");  
  
        if (strcmp(mazeStyle, "bombed") == 0) {  
            _instance = new BombedMazeFactory;  
  
        } else if (strcmp(mazeStyle, "enchanted") == 0) {  
            _instance = new EnchantedMazeFactory;  
  
            // ... other possible subclasses  
  
        } else { // default  
            _instance = new MazeFactory;  
        }  
    }  
    return _instance;  
}
```

# Factory

# Factory Method

## Object Diagram for *Factory Method* using Injection Mold 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, etc.) is determined by the mold

# Factory Method

## 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.

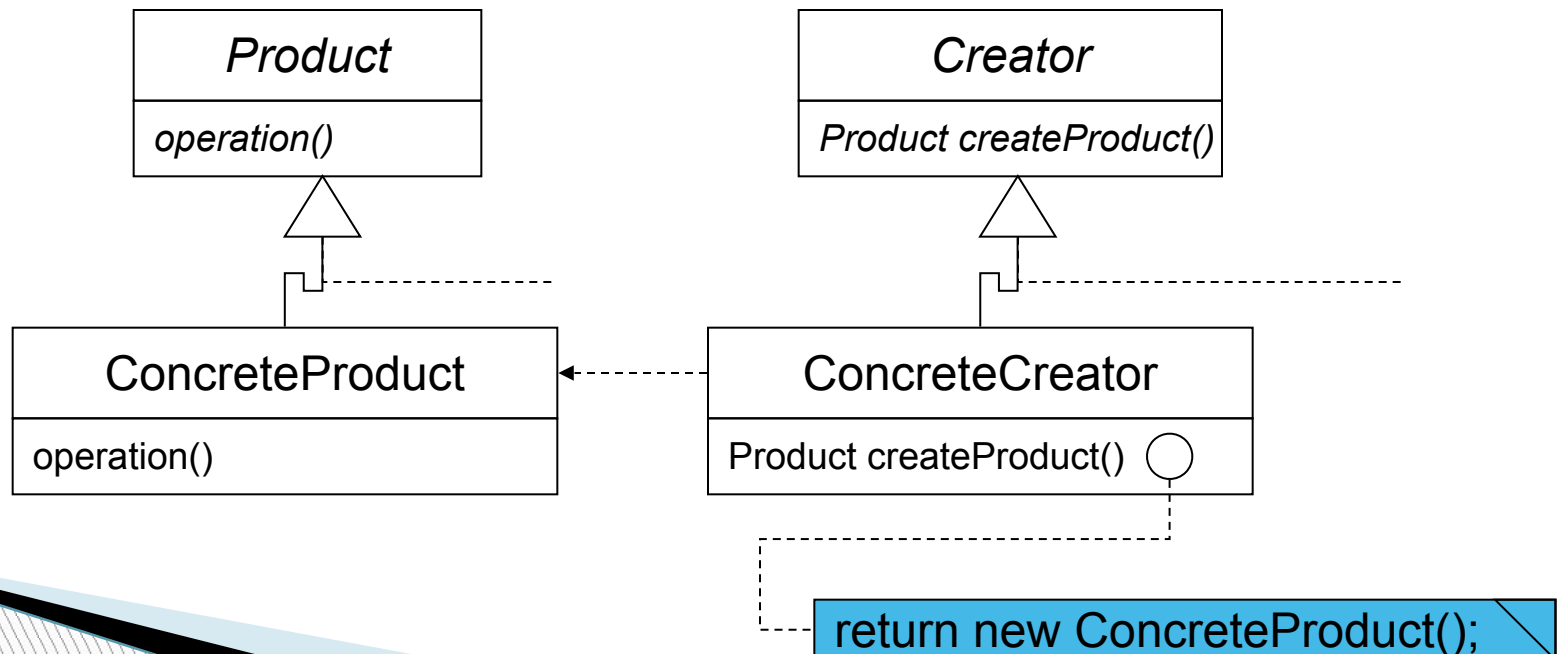
## Also Known As

Virtual Constructor



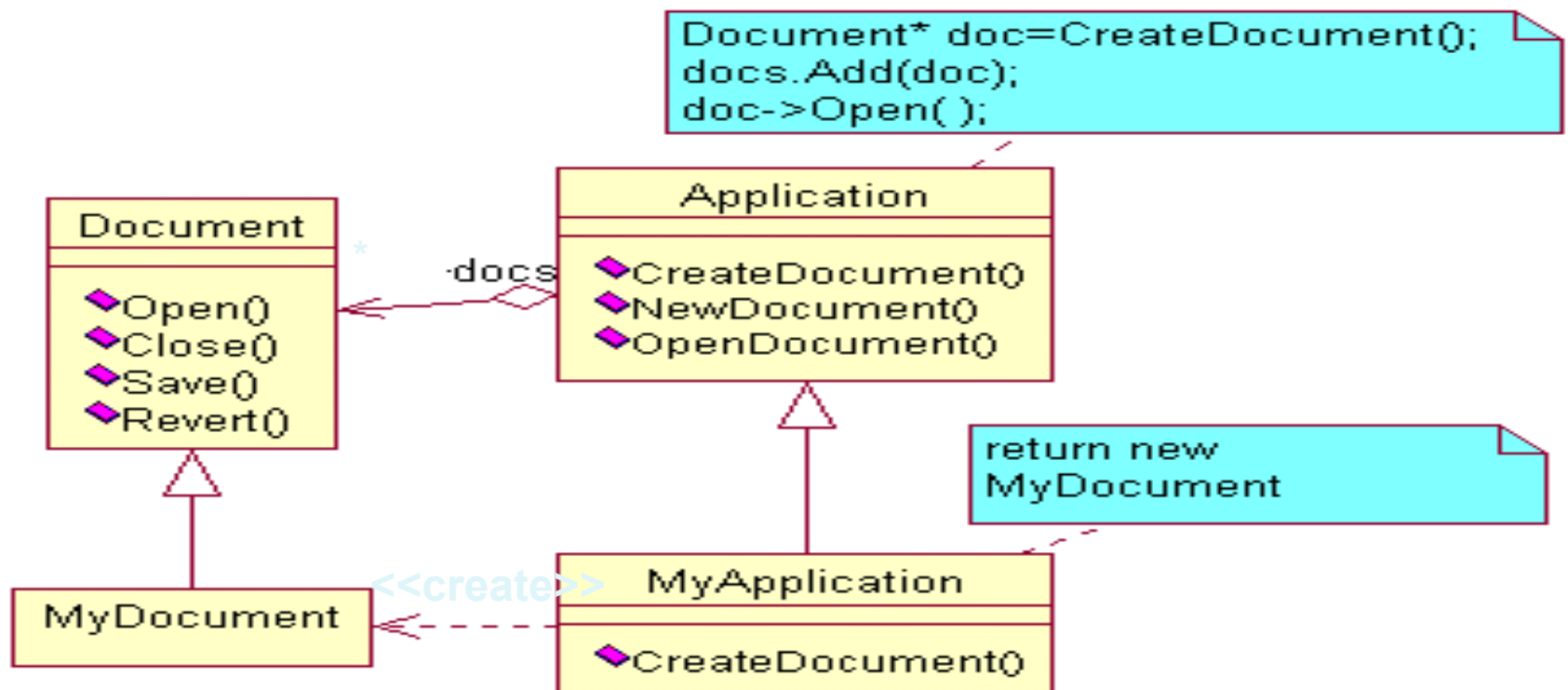
# Factory Method

- ▶ Defer object instantiation to subclasses
- ▶ Eliminates binding of application-specific subclasses
- ▶ Connects parallel class hierarchies
- ▶ A related pattern is AbstractFactory



# Problem :

“A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation”





# Illustration: Example

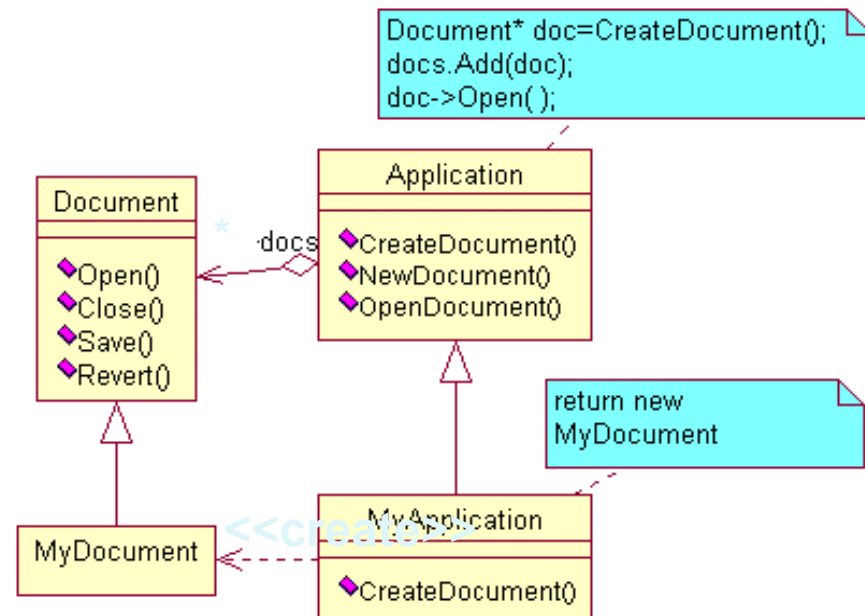
## Application that can present multiple documents to the user

Frameworks use abstract classes to define and maintain relationships between objects.

A framework is often responsible for creating these objects as well.

**As shown in the figure :**

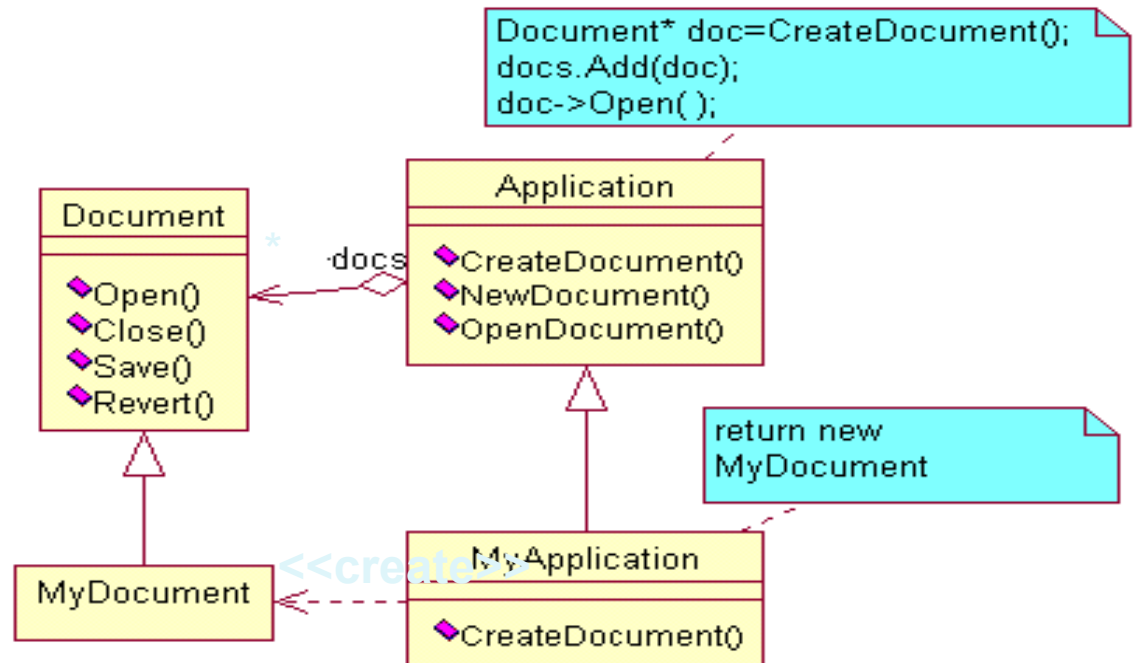
Application and Document which form a relationship need to be extended by the client applications



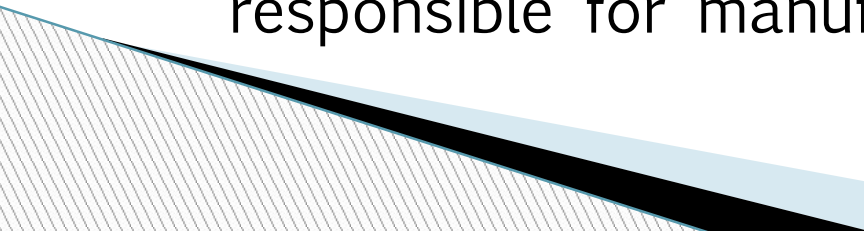
## Illustration :

Because the particular document subclass to instantiate is application specific, the application class can't predict the subclass of the document to instantiate.

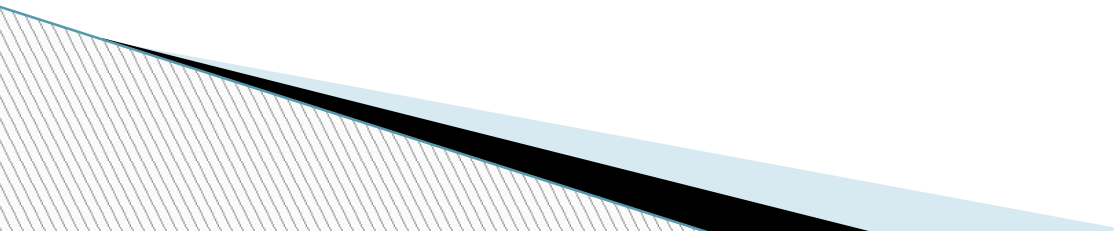
**The problem here is :** The framework must instantiate classes, but it only knows about abstract classes, which it can not instantiate



## Illustration : Solution

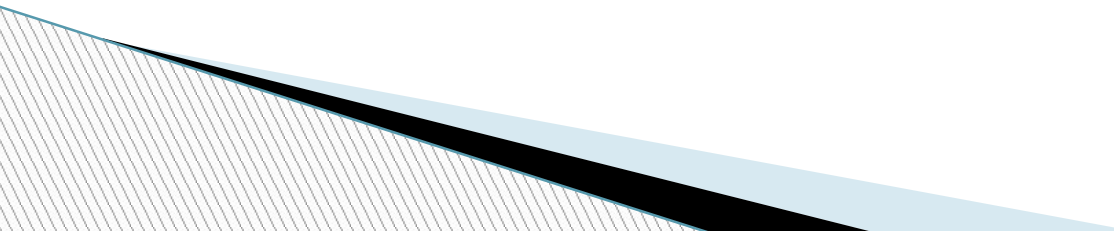
- ▶ ***Factory Method Pattern* offer a solution. It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.**
  - ▶ Application subclasses redefine an abstract `CreateDocument( )` on Application to return an appropriate Document subclass.
  - ▶ Once an application subclass is instantiated , It can then instantiate application – specific Documents without knowing their Class , We call `CreateDocument( )` a Factory Method because it is responsible for manufacturing an object.
- 

# Solution :

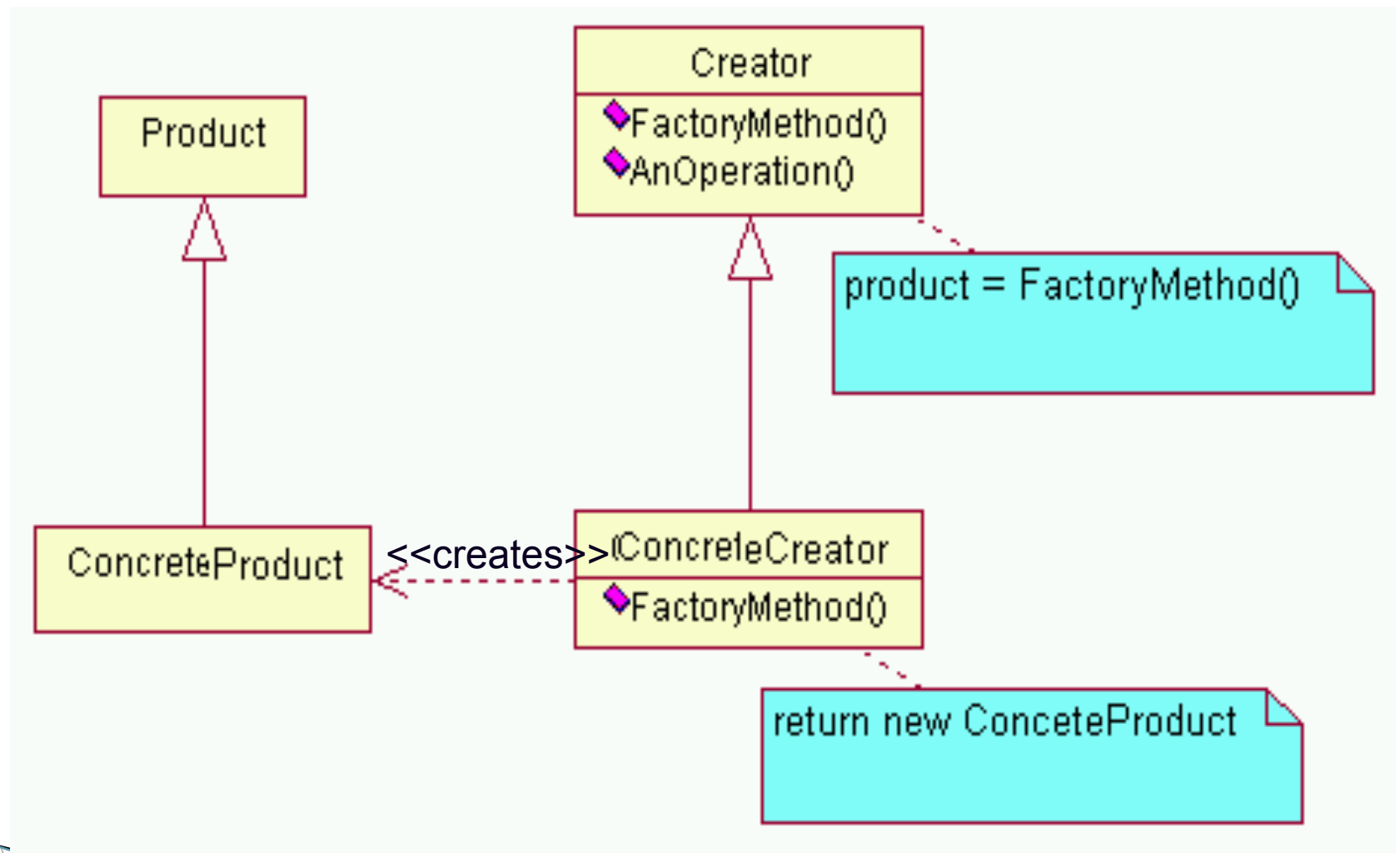
- ▶ A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.
  - ▶ Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.
- 

# Applicability

## Use the Factory Method pattern when

- ▶ A class can't anticipate the class objects it must create.
  - ▶ A class wants its subclasses to specify the objects it creates.
  - ▶ Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.
- 

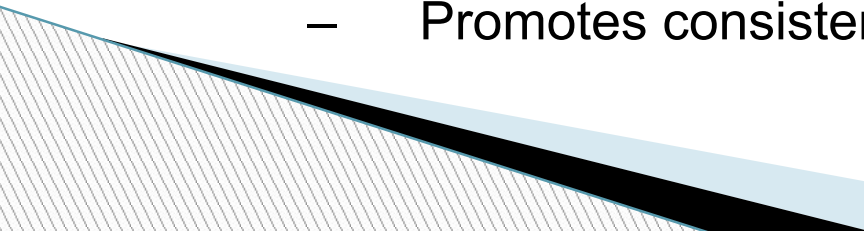
# Structure



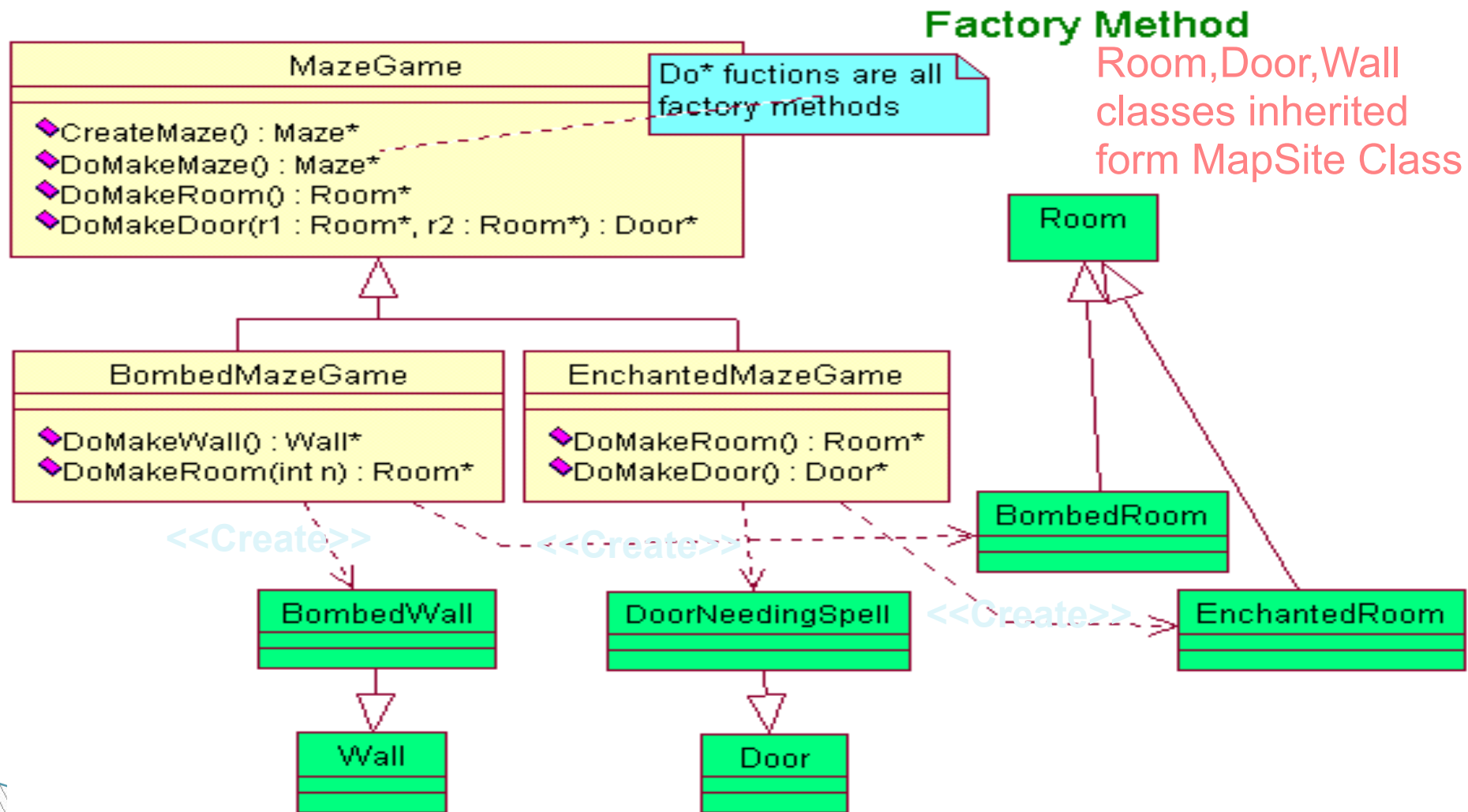
# Collaborations

- ▶ Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

## Consequences

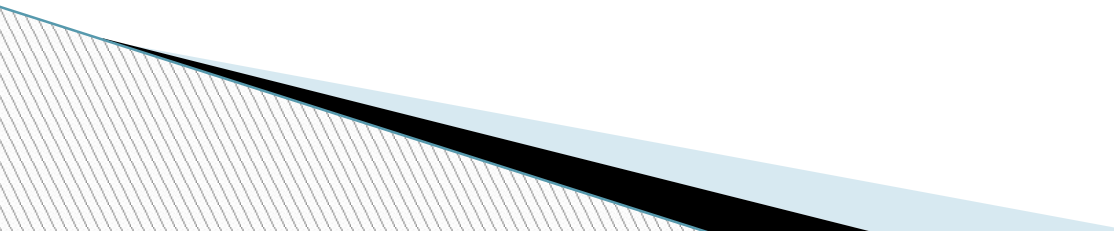
- Connects parallel class hierarchies.
    - Isolates concrete classes
    - Makes exchanging product families easy
    - Promotes consistency among products
- 

# Model



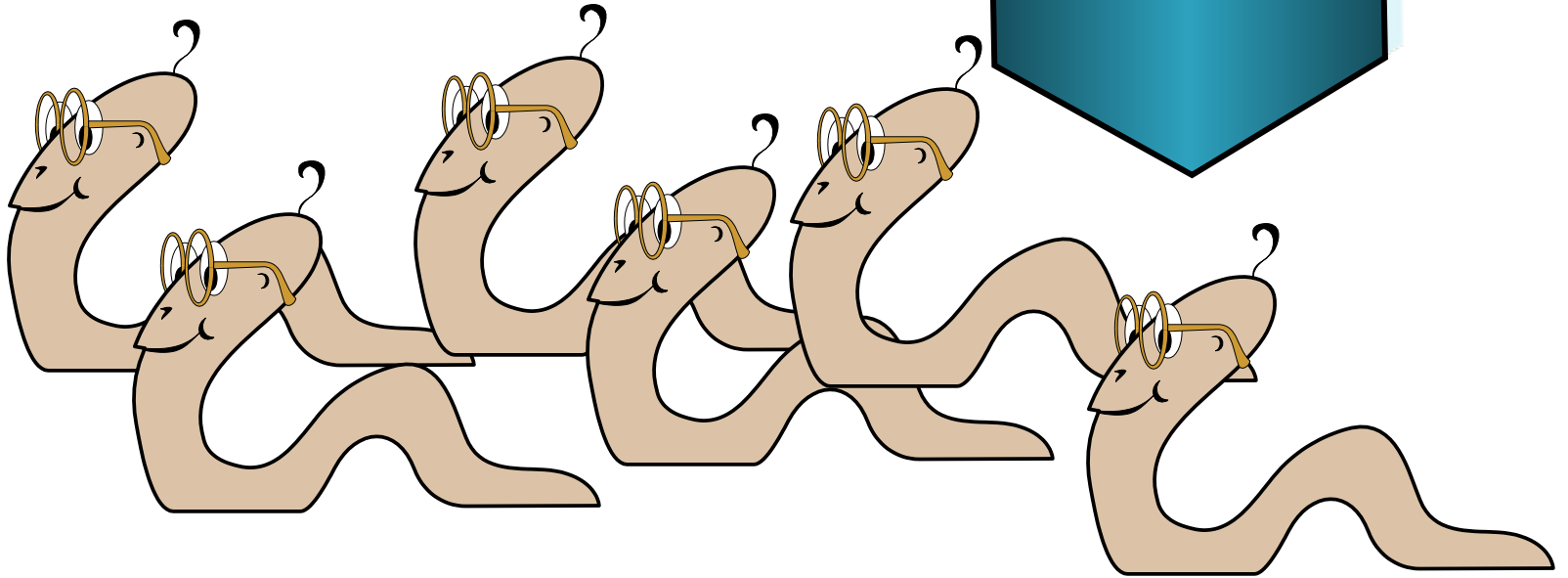
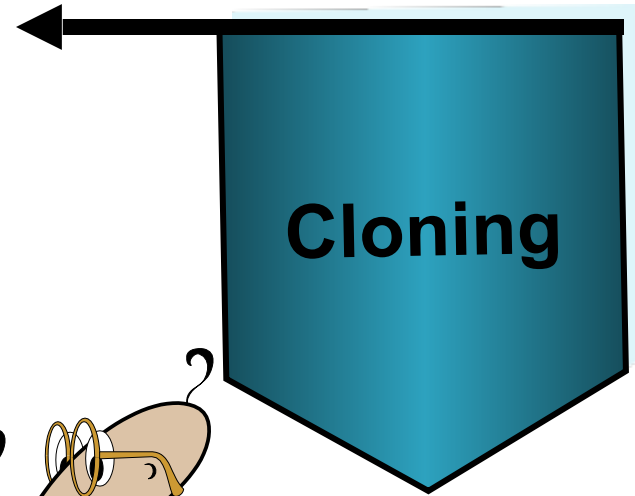
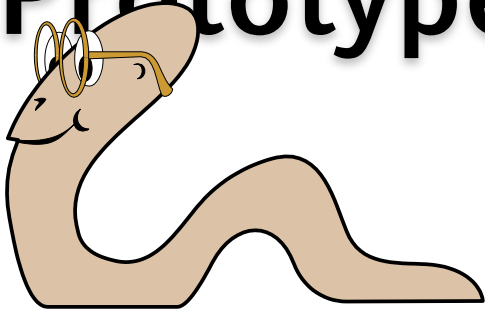


## Related Patterns

- ▶ Abstract factory is often implemented with factory methods.
  - ▶ Factory Method is to creating objects as Template Method is to implementing an algorithm
  - ▶ Factory Methods are usually called within Template Methods.
  - ▶ Factory Method: creation through inheritance.  
Prototype: creation through delegation
- 

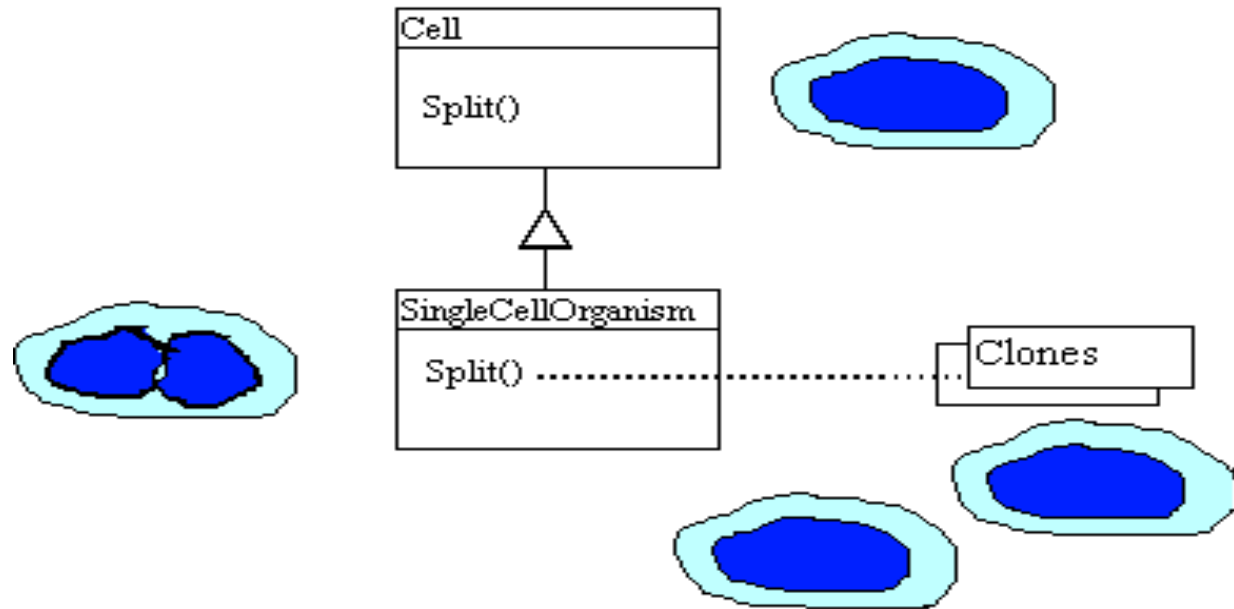
# Prototype

# Prototype



# Prototype

Object Diagram for  
*Prototype* using Cell  
Division Example



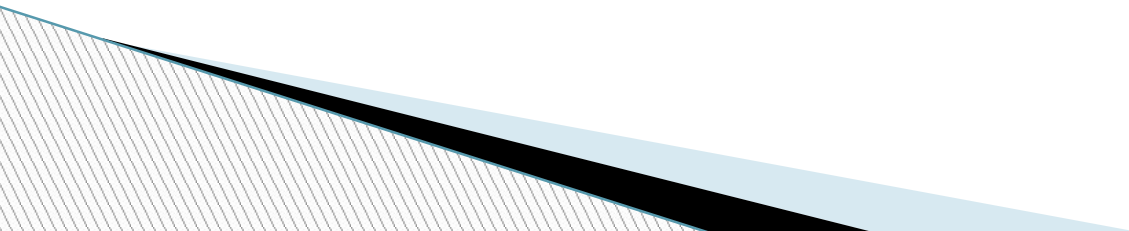
The *Prototype* pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive, and does not participate in copying itself. The mitotic division of a cell, resulting in two 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.

# Intent

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

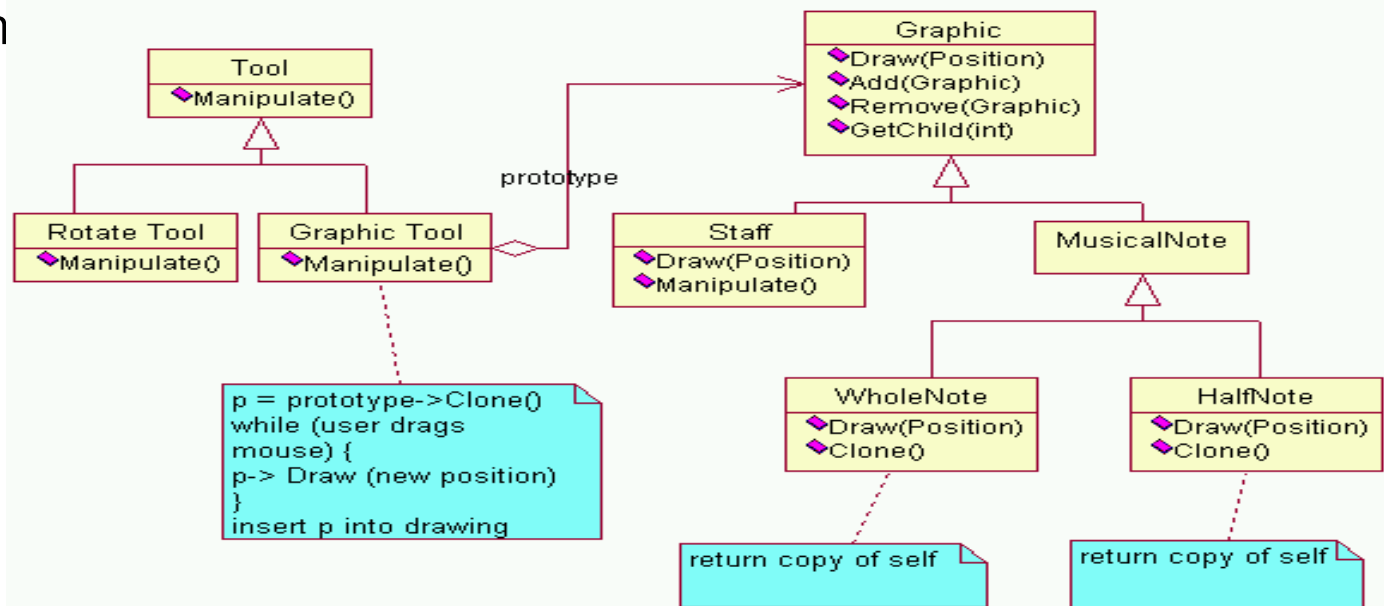
## Problem:

Application "hard wires" the class of object to create in each "new" expression



# Illustration:

- ▶ In the figure the framework provides an abstract Graphic class for graphical components, like notes and staves. Moreover it will provide an abstract tool class for defining tools like those in the palette.
- ▶ The framework also predefines a GraphicTool subclass for tools that create instances of graphical objects and add them to the document

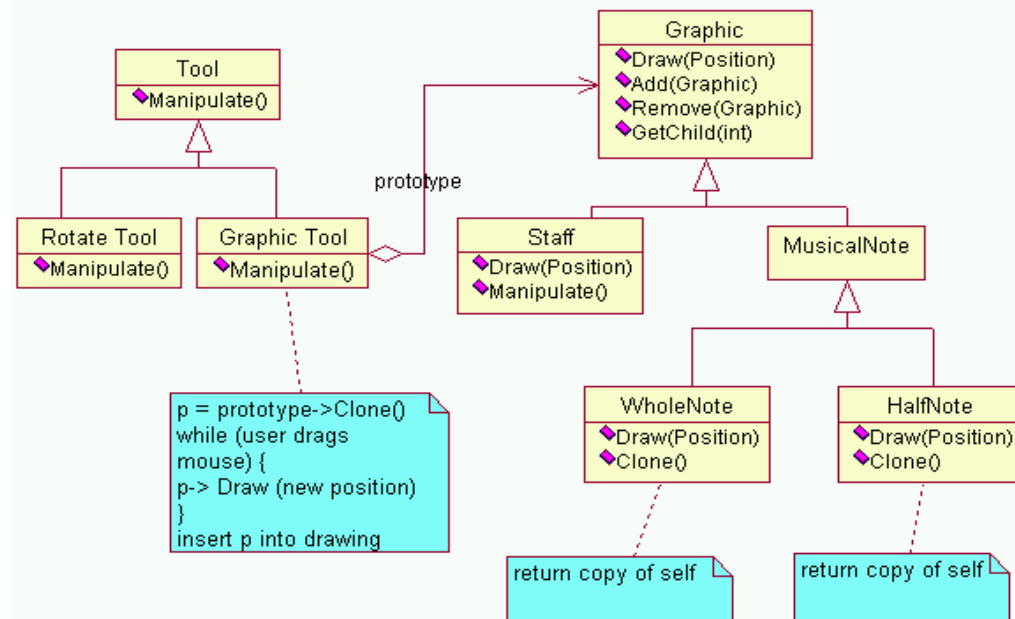


## Illustration:Problem

Here the **Problem is** the GraphicTool, ie the classes for notes and staves are specific to our application, but the GraphicTool class belongs to the framework. The GraphicTool doesn't know how to create instances of our music classes to add to the score.

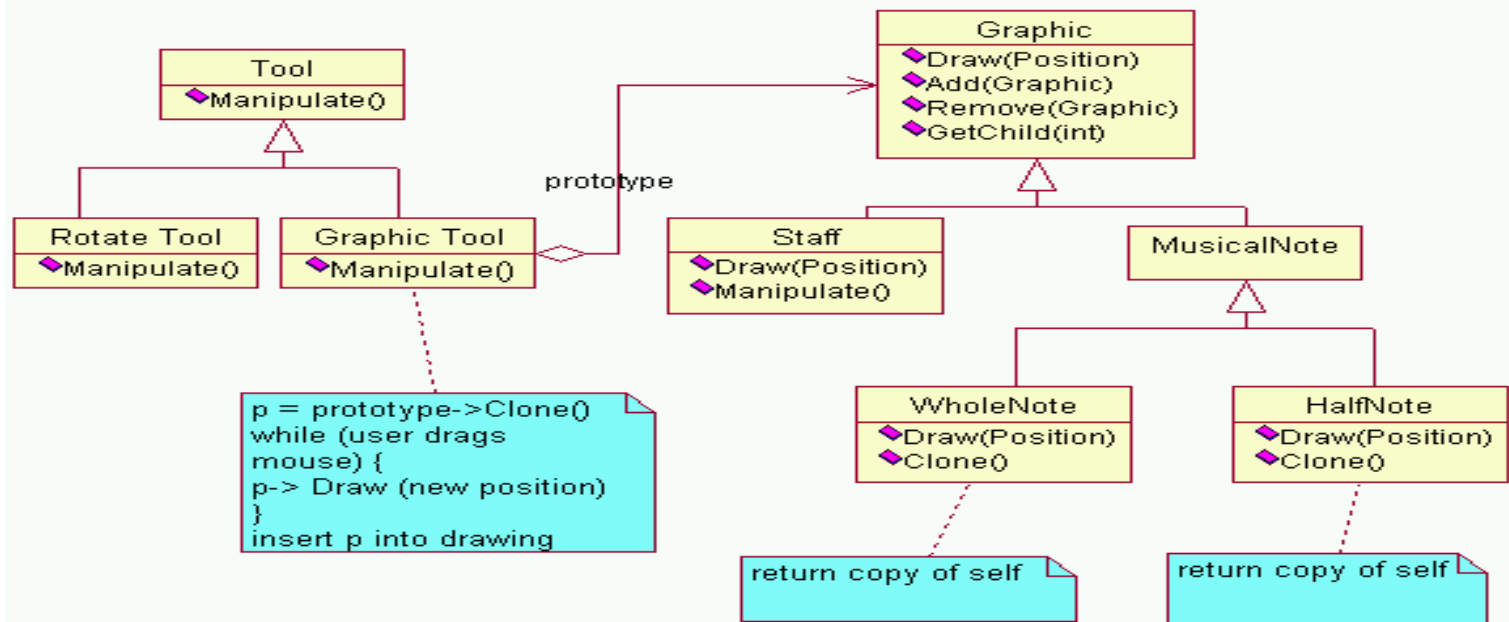
The objective is to minimize the height of the Client (GraphicTool) Class Hierarchy given the prototype hierarchy.

The question is , how can the framework use the Graphic to parameterize instances of GraphicTool by the class of Graphic they're supposed to create?



## Illustration: Solution

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.

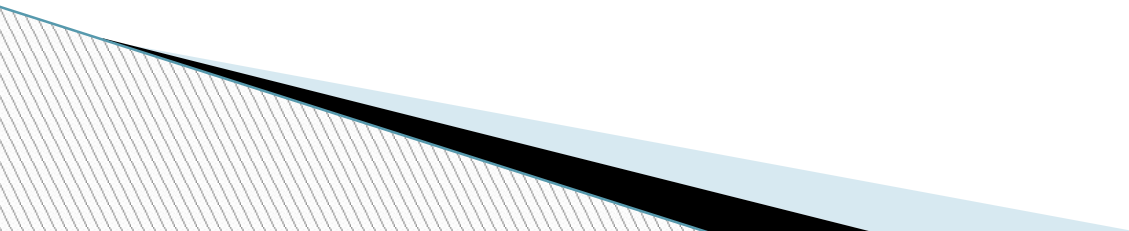




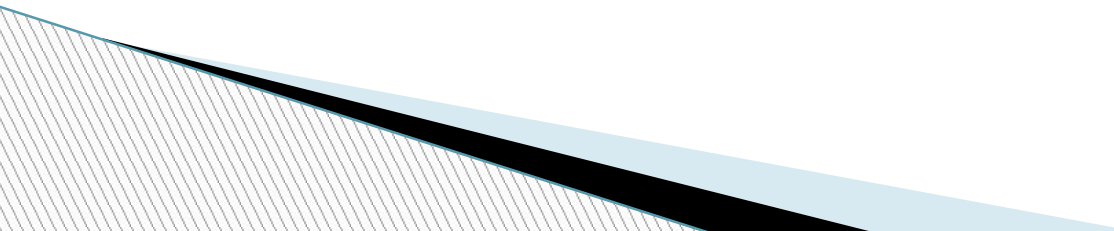
# Solution

Declare an abstract base class that specifies a pure virtual "clone" method, and, maintains a dictionary of all "cloneable" concrete derived classes. Any class that needs a "polymorphic constructor" capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

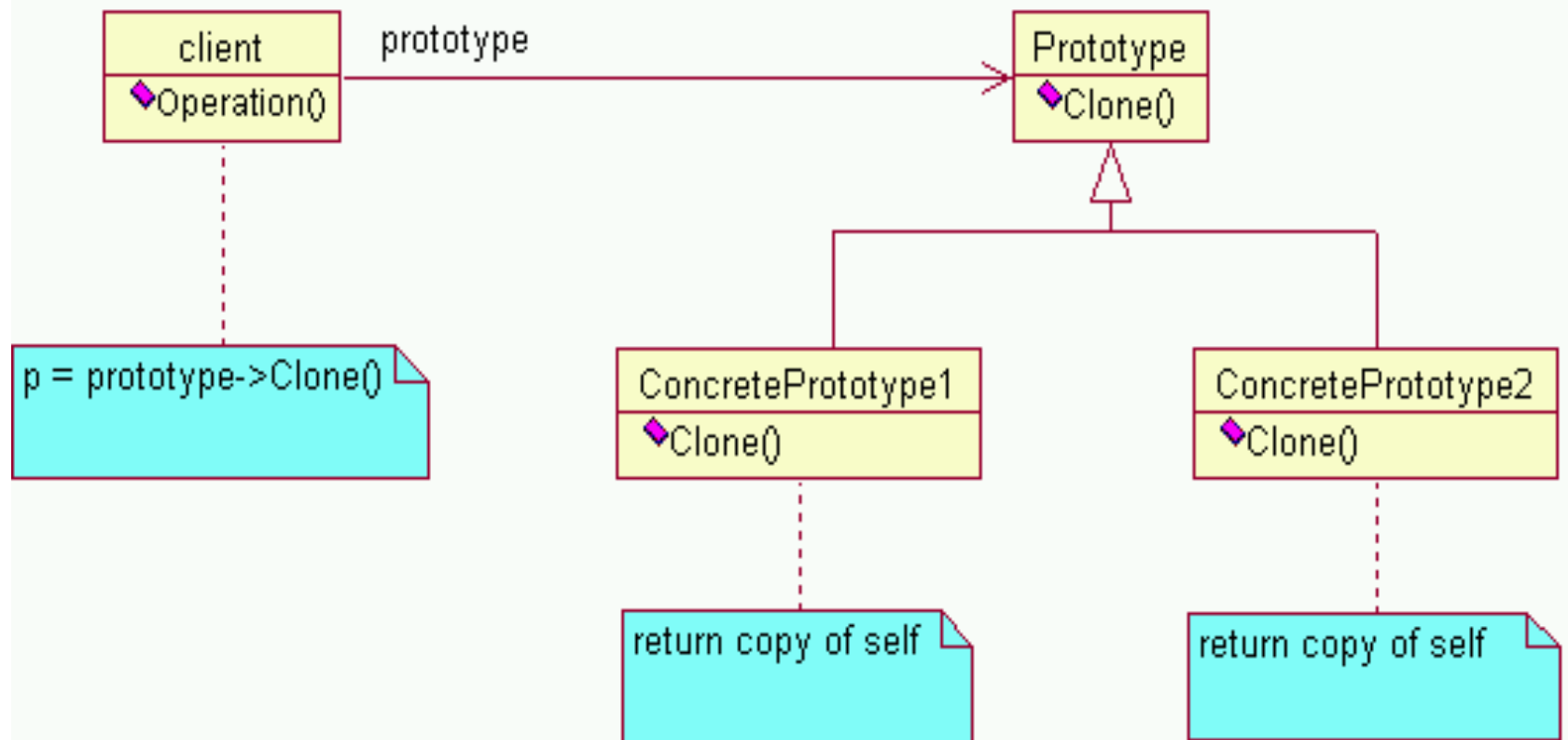
The client then, instead of writing code that invokes the "new" operator on a hard-wired class name, calls a "clone" operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.



## Applicability

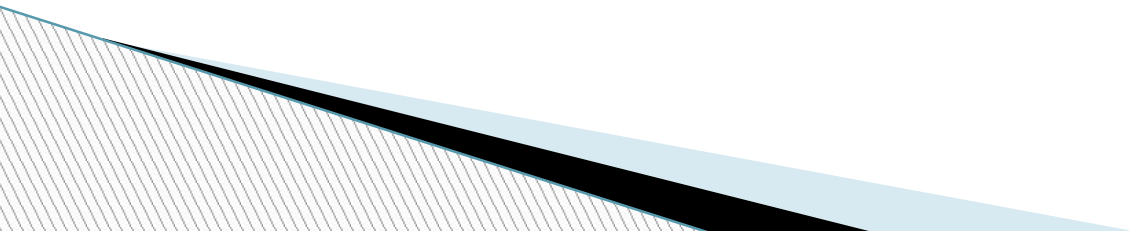
- ▶ Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and
  - ▶ When the classes to instantiate are specified at run-time, for example, by dynamic loading; or
  - ▶ To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
  - ▶ When instances of a class can have one of only a few different combinations of state. 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

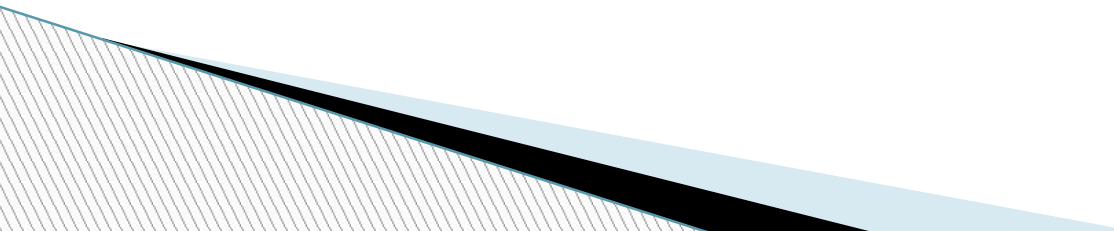


# Collaborations

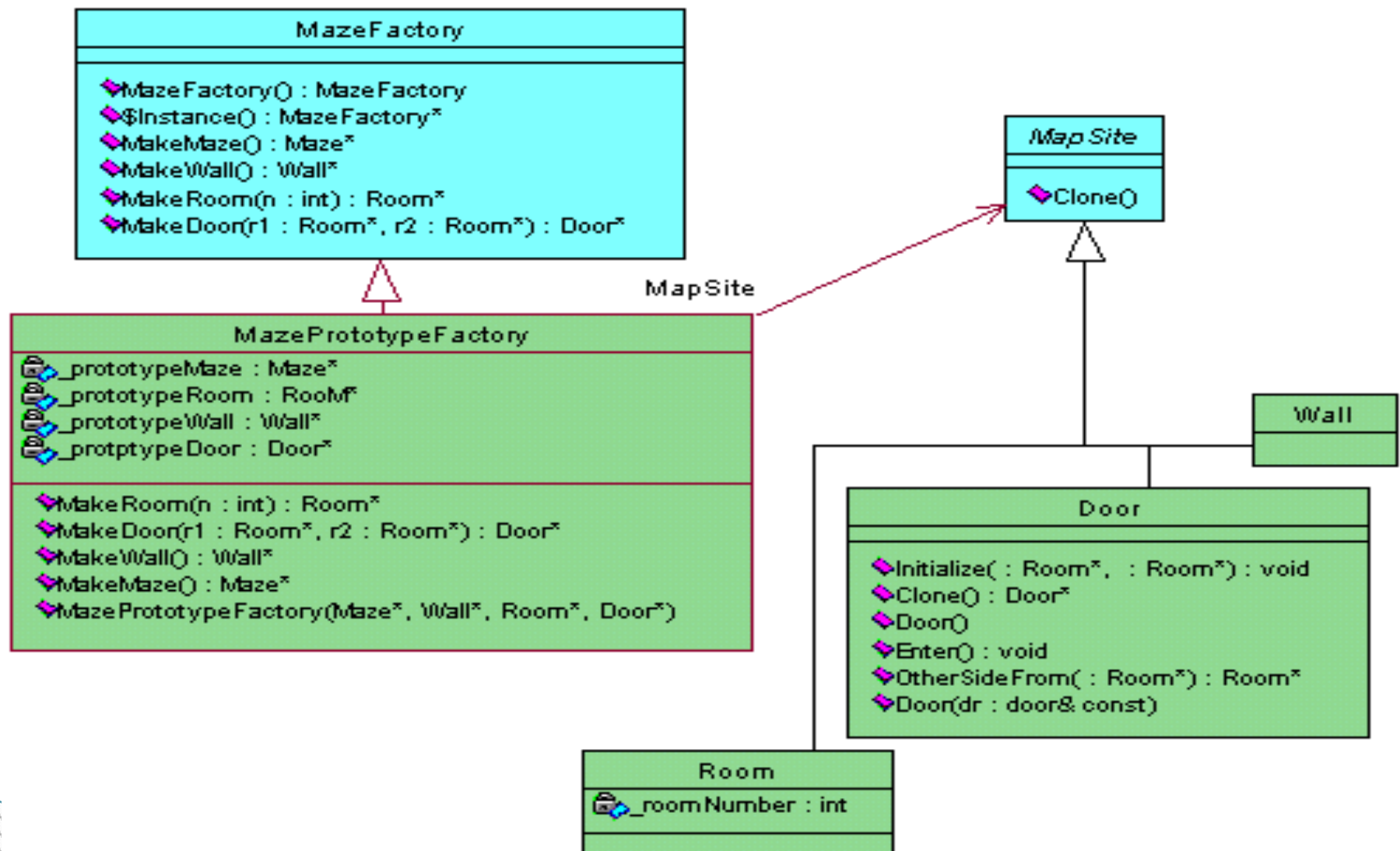
- ▶ A client asks a prototype to clone itself.



## Consequences

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

# Model

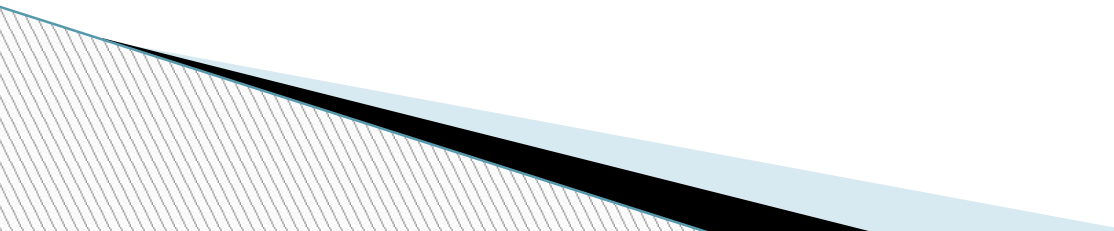


## code

```
MazePrototypeFactory::MazePrototypeFactory (  
    Maze* m, Wall* w, Room* r, Door* d  
) {  
    _prototypeMaze = m;  
    _prototypeWall = w;  
    _prototypeRoom = r;  
    _prototypeDoor = d;  
}  
Wall* MazePrototypeFactory::MakeWall () const {  
    return _prototypeWall->Clone();  
}  
Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {  
    Door* door = _prototypeDoor->Clone();  
    door->Initialize(r1, r2);  
    return door;  
}
```


## code

```
Door::Door (const Door& other) {  
    _room1 = other._room1;  
    _room2 = other._room2;  
}  
void Door::Initialize (Room* r1, Room* r2)  
{  
    _room1 = r1;  
    _room2 = r2;  
}  
Door* Door::Clone () const {  
    return new Door(*this);  
}
```





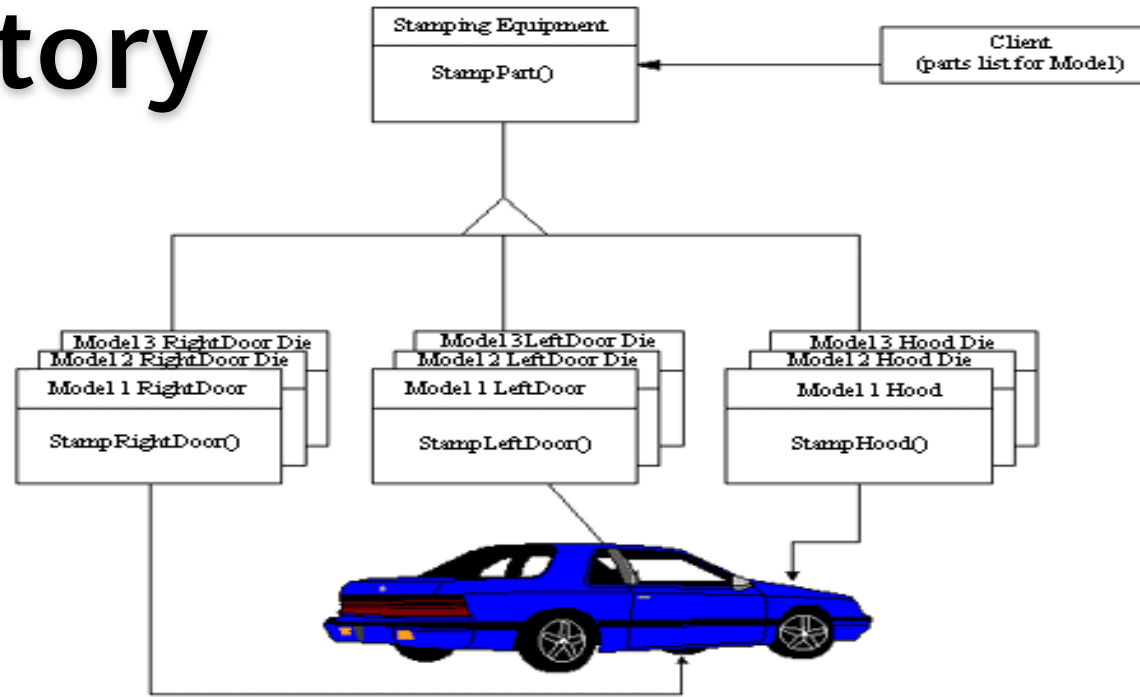
## Related Patterns

- ▶ Prototype and Abstract Factory are completing patterns in some ways, as we discuss at the end of this chapter. 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.
- 

# Abstract Factory

# Abstract Factory

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.



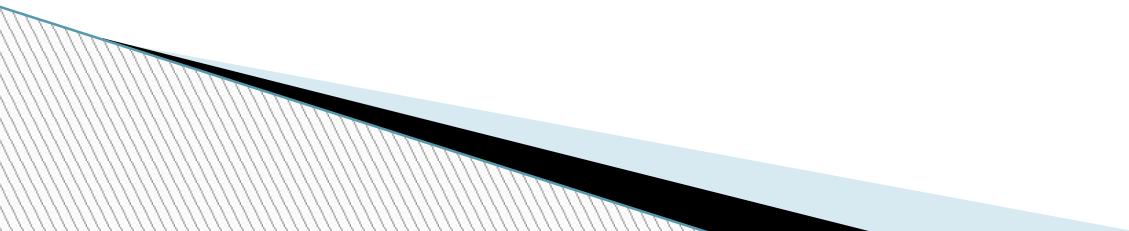
The stamping equipment is an *Abstract Factory* which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes

# Intent

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

## Alternative Name:

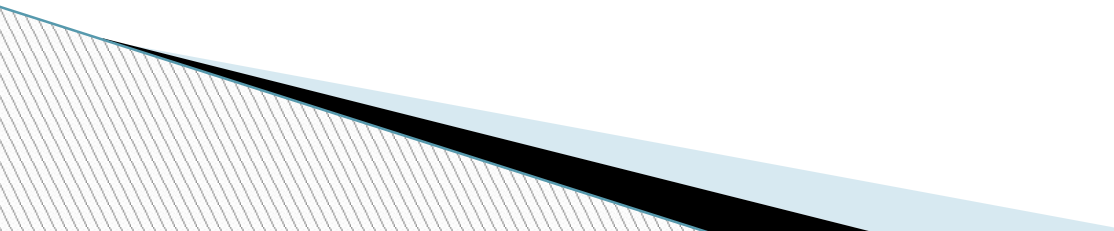
- Kit



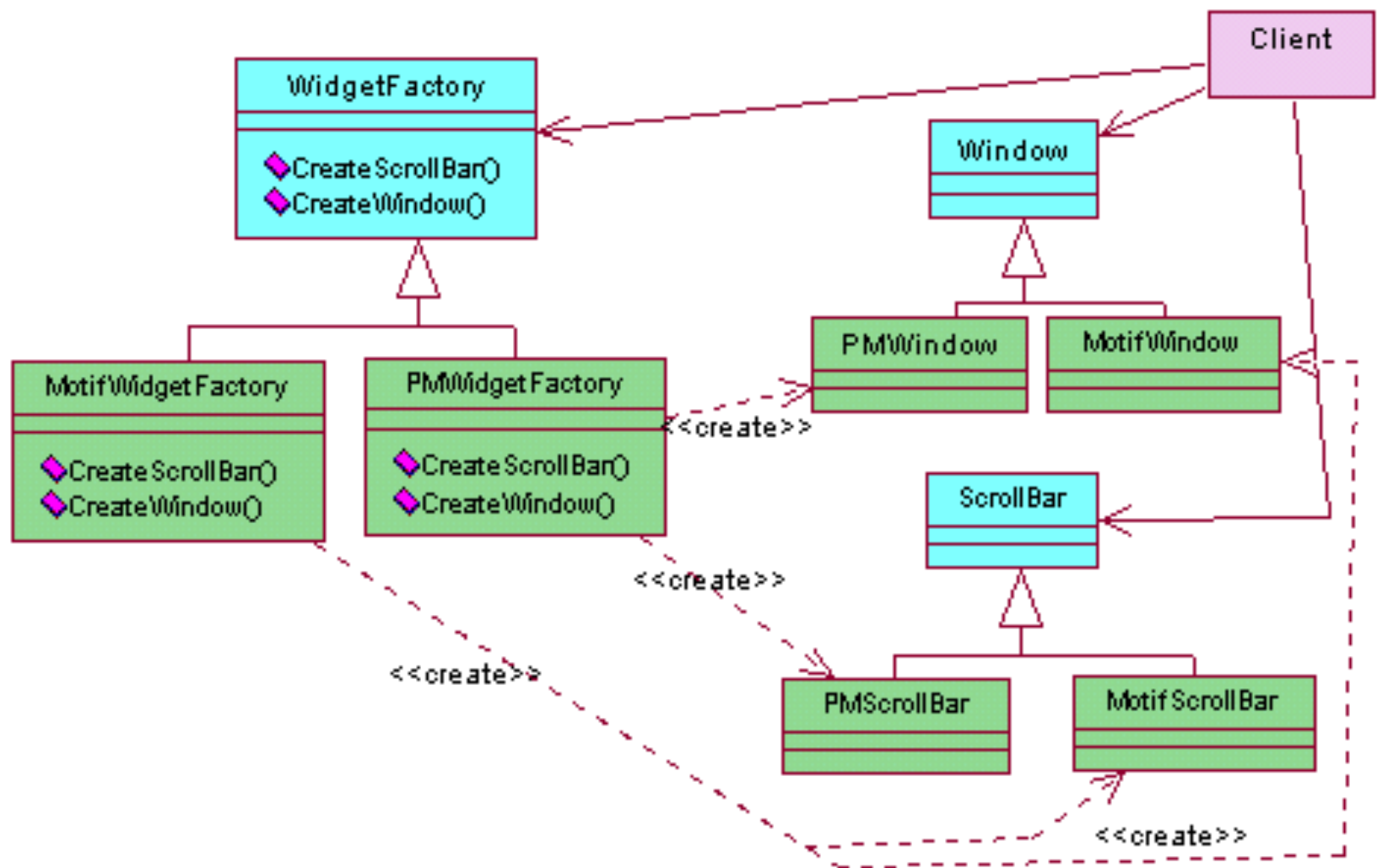
## **Problem:**

- ▶ How should one go about designing applications which have to be adopted to different Look and feel standards?

## **Solution:**

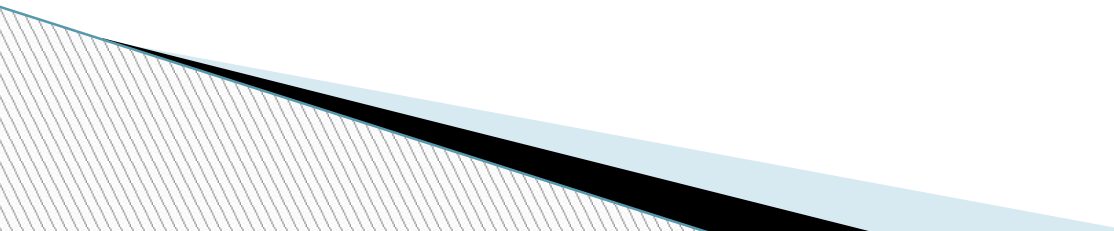
- ▶ One could solve this problem by defining abstract widgets factory class that declares an interface for creating each basic kind of widgets ( control ).
  - ▶ There is also an abstract class for each kind of widget, and concrete sub classes implement widgets for specific look and standards.
- 

# Illustration

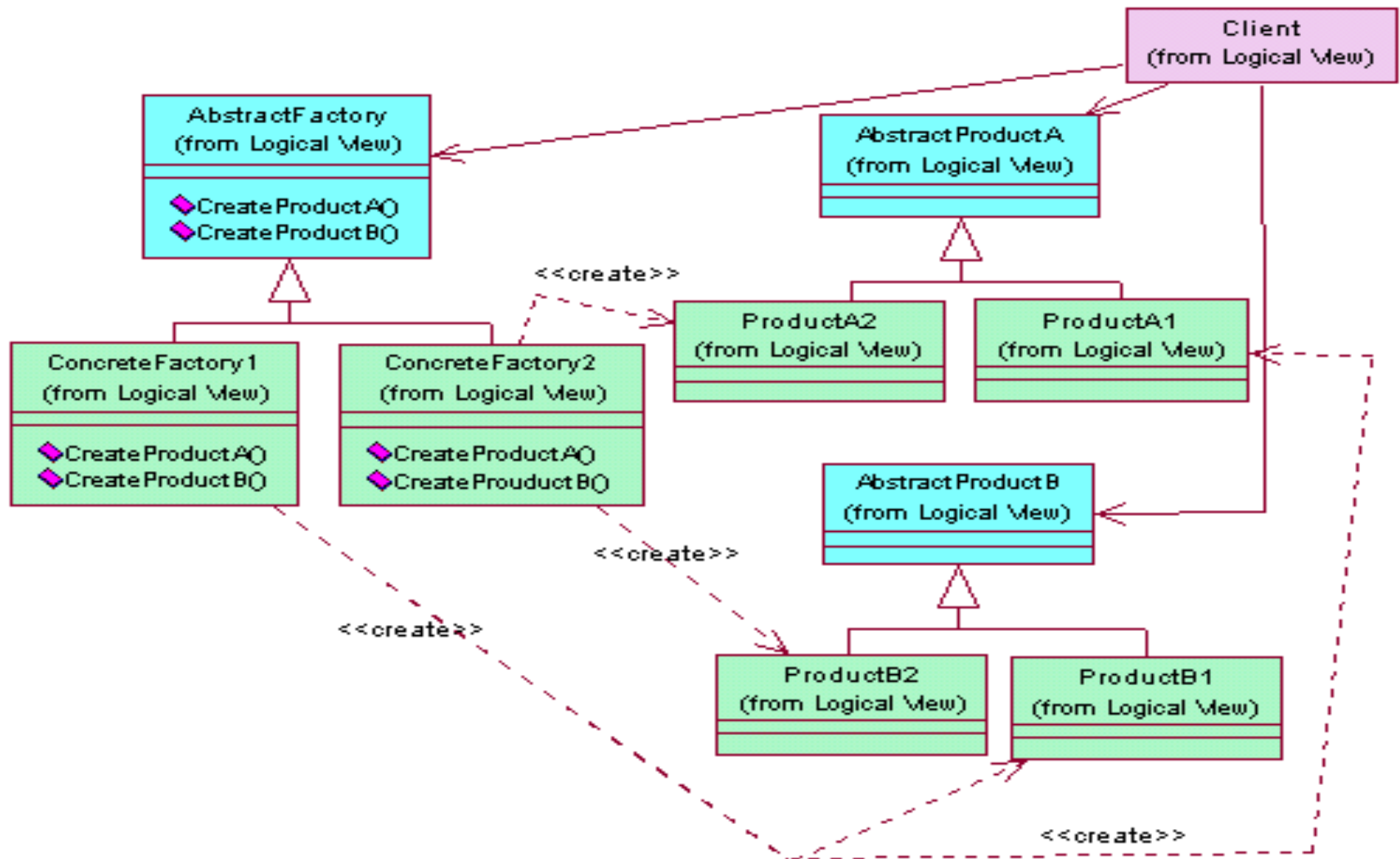


# Applicability

## Use the Abstract Factory pattern when

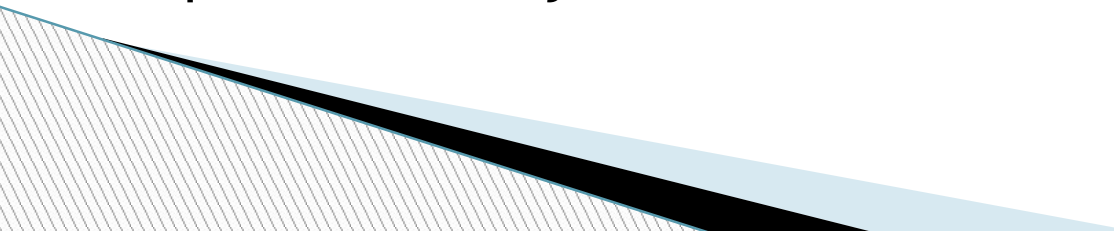
- ▶ A system should be independent of how its products are created, composed, and represented.
  - ▶ A system should be configured with one of multiple families of products.
  - ▶ A family of related product objects is designed to be used together, and you need to enforce this constraint.
  - ▶ You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
- 

# Structure



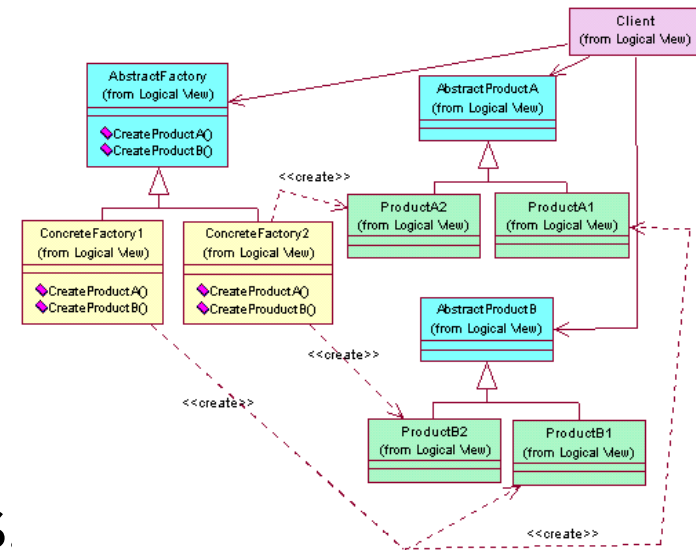


# Collaborations

- Normally a single instance of a ConcreteFactory 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.
  - AbstractFactory defers creation of product objects to its ConcreteFactory subclass.
- 

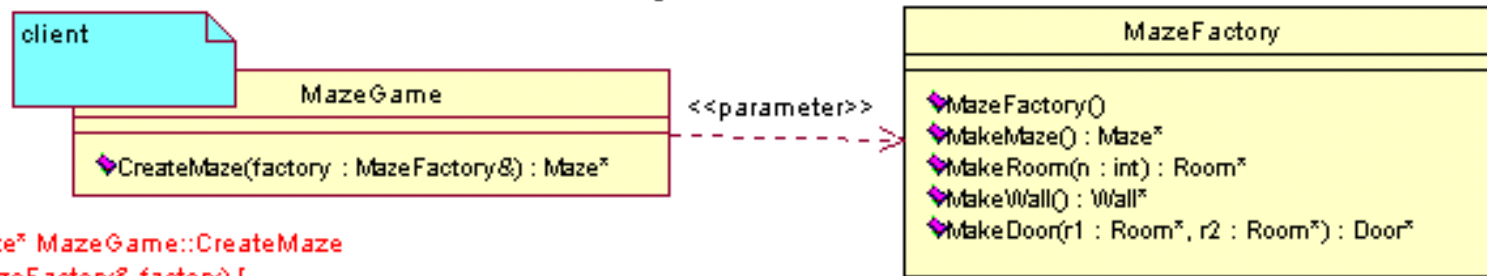
# Consequences

- ▶ It isolates concrete classes.
- ▶ It makes exchanging product families easy.
- ▶ It promotes consistency among products.
- ▶ Supporting new kinds of products is difficult.



# Model

## Abstract Factory

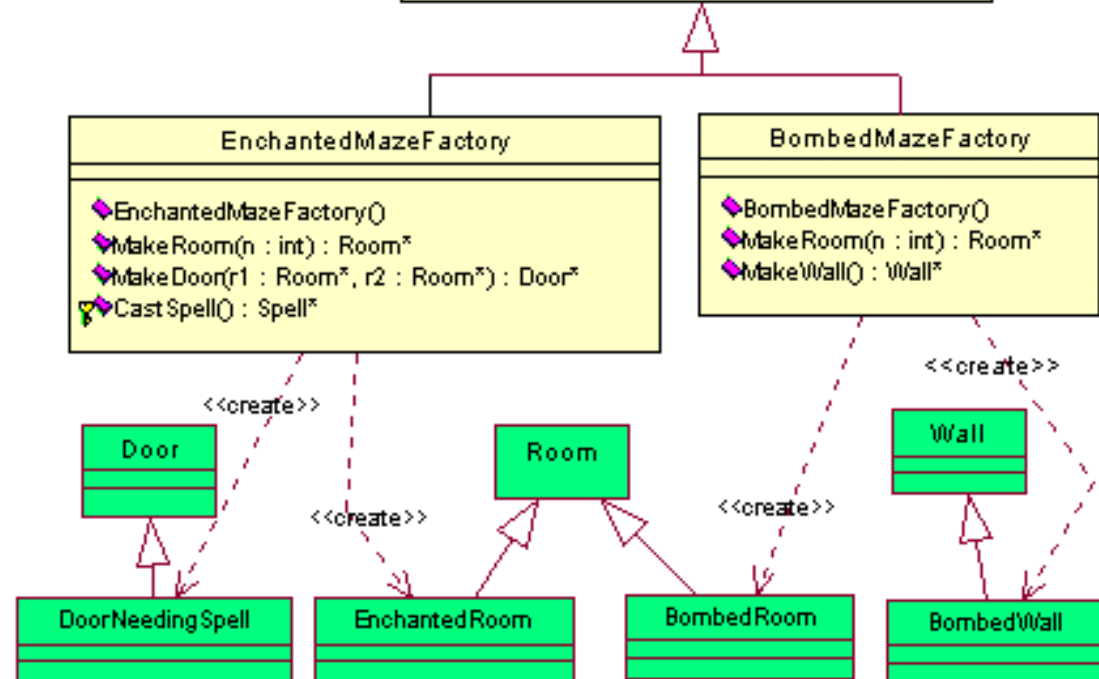


```
Maze* MazeGame::CreateMaze
(MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);
```

```
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
}

r1->SetSide(North, factory.MakeWall());
r1->SetSide(East, aDoor);
r1->SetSide(South, factory.MakeWall());
r1->SetSide(West, factory.MakeWall());

r2->SetSide(North, factory.MakeWall());
r2->SetSide(East, factory.MakeWall());
r2->SetSide(South, factory.MakeWall());
r2->SetSide(West, aDoor);
```



# Related Patterns

- ▶ AbstractFactory classes are often implemented with factory methods, but they can also be implemented using Prototype.

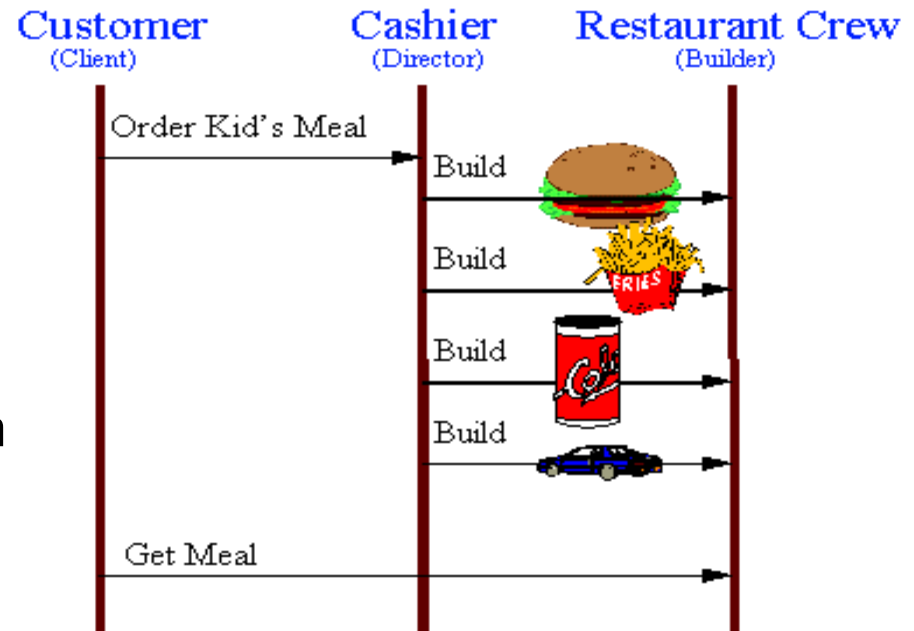
# Builder

# BUILDER

Object Interaction Diagram for the *Builder* using Kid's Meal 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, etc ). 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, etc . 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.

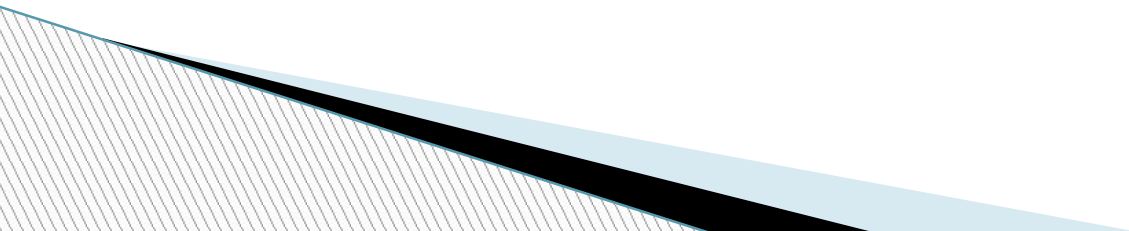


## **Intent**

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

## **Problem**

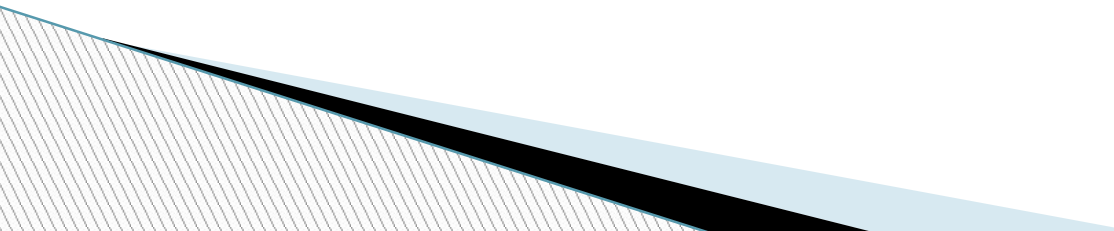
- ▶ An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage



**Illustration:** Example document exchange format.

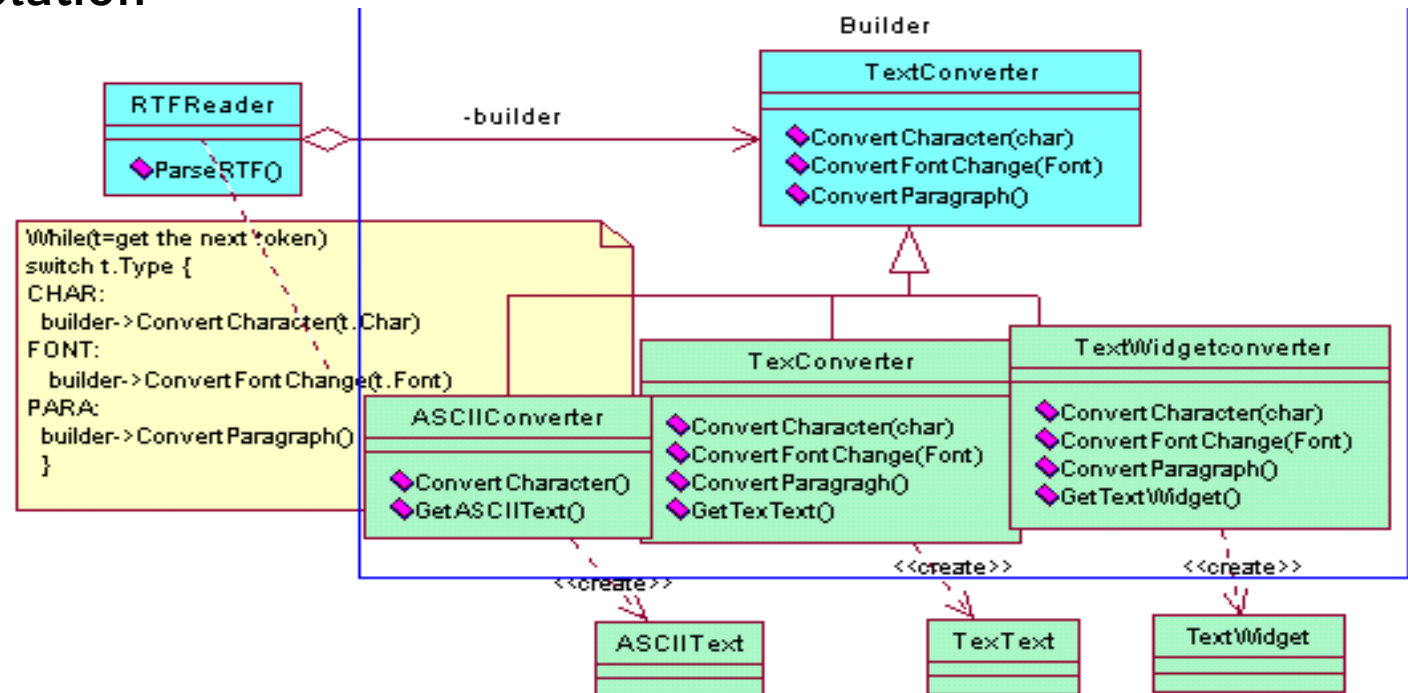
Lets say RTF ( Rich Text Format) document exchange format should be able to be convert RTF to many text formats. The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively.

The **problem**, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.





## Illustration: Solution



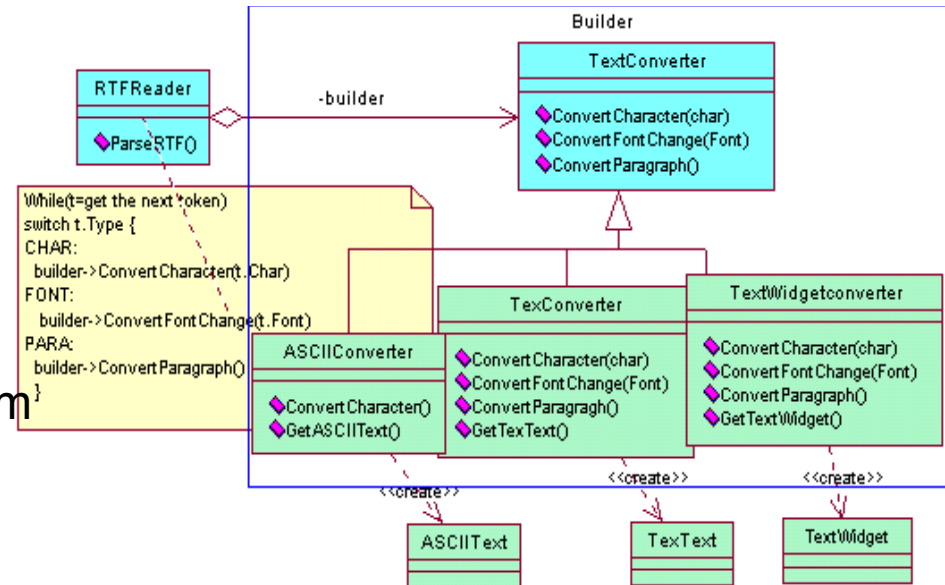
A solution is to configure the **RTFReader** class with **TextConverter** object that converts RTF to another textual representation. As the **RTFReader** parses the RTF document, it uses the **TextConverter** to perform the conversion. Whenever the **RTFReader** recognizes an RTF token (either plain text or an RTF control word), it issues a request to the **TextConverter** to convert the token. **TextConverter** objects are responsible both for performing the data conversion and for representing the token in a particular format.

# Illustration: Solution

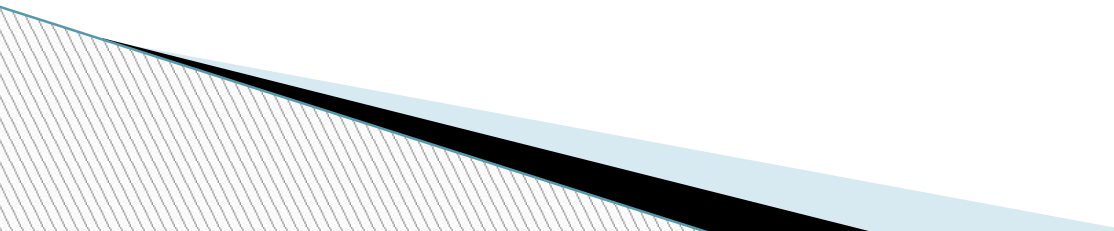
Each converter classes like ASCIIConverter, TexConverter, TextWidgetConverter takes the mechanism for creating and assembling a complex object and puts it behind an abstract interface.

The converter is separate form the reader,

Each converter class is called **builder** in the pattern, and the reader is called **director**.

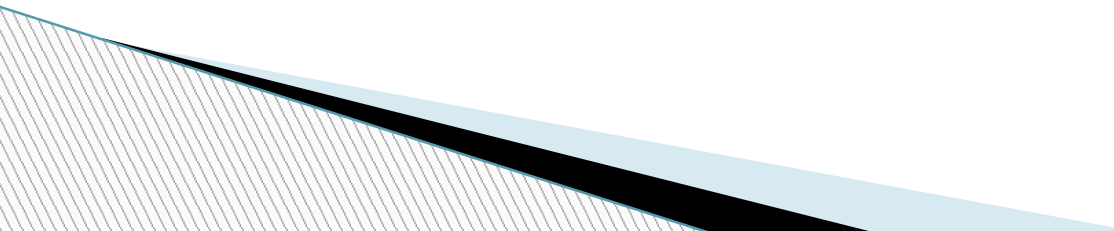


## Solution:

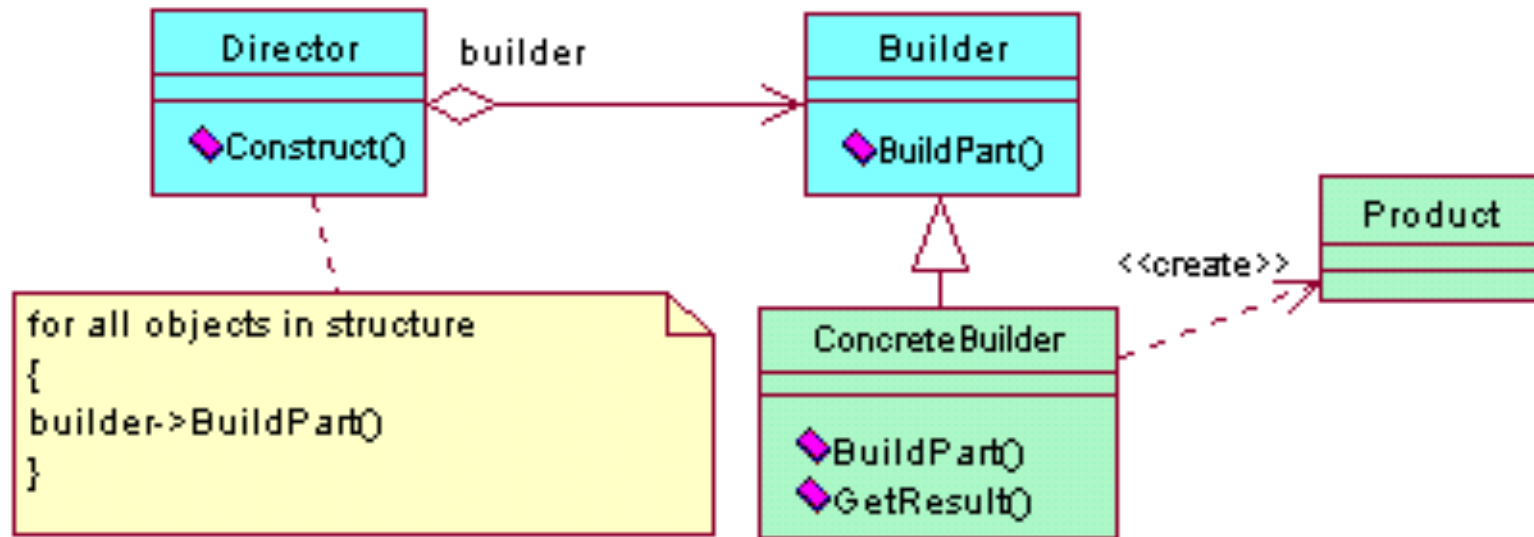
- ▶ Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget).
  - ▶ The focus/distinction is on creating complex aggregates.
  - ▶ The "director" invokes "builder" services as it interprets the external format. The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".
  - ▶ Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".
- 

Applicability

## **Use the Builder pattern when**

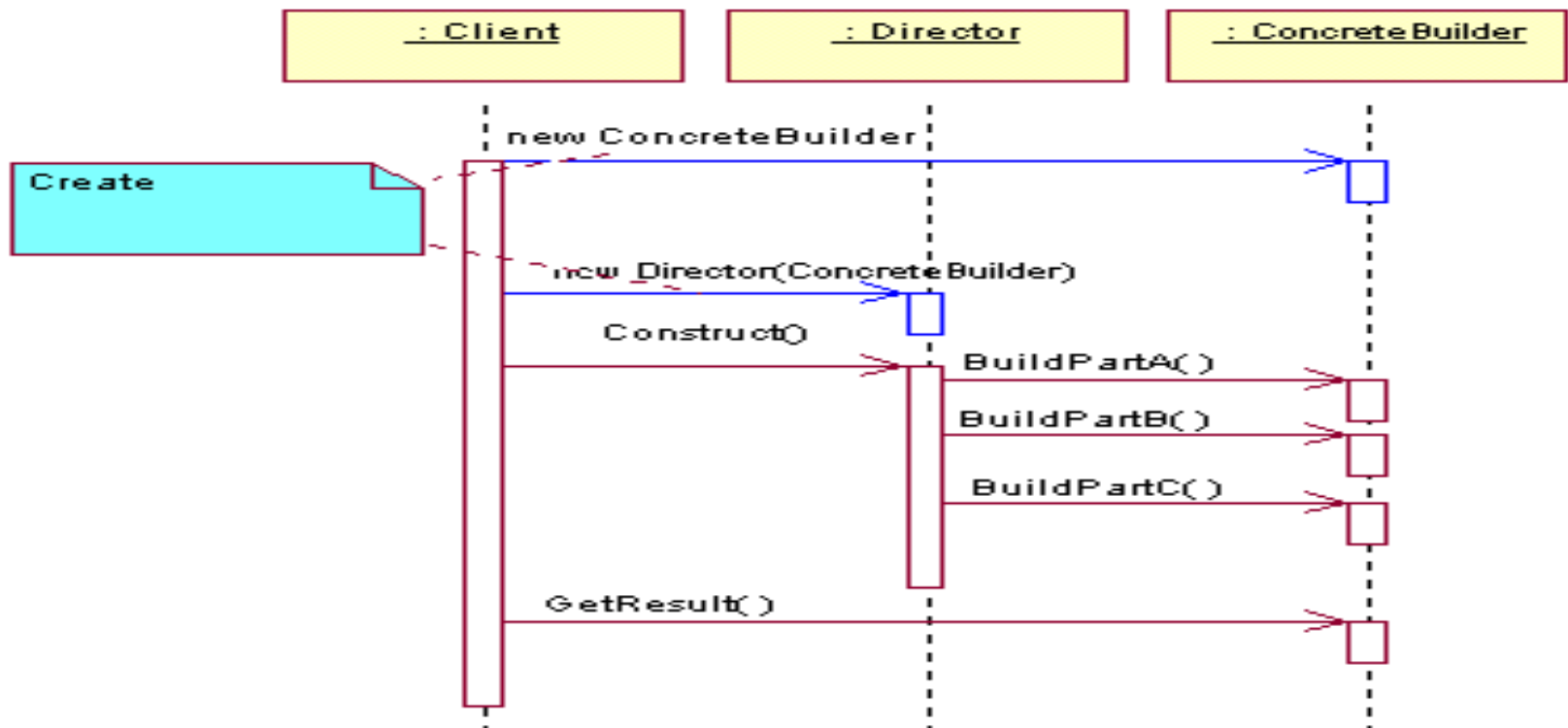
- ▶ the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
  - ▶ The construction process must allow different representations for the object that's constructed.
- 

# Structure



# Collaborations

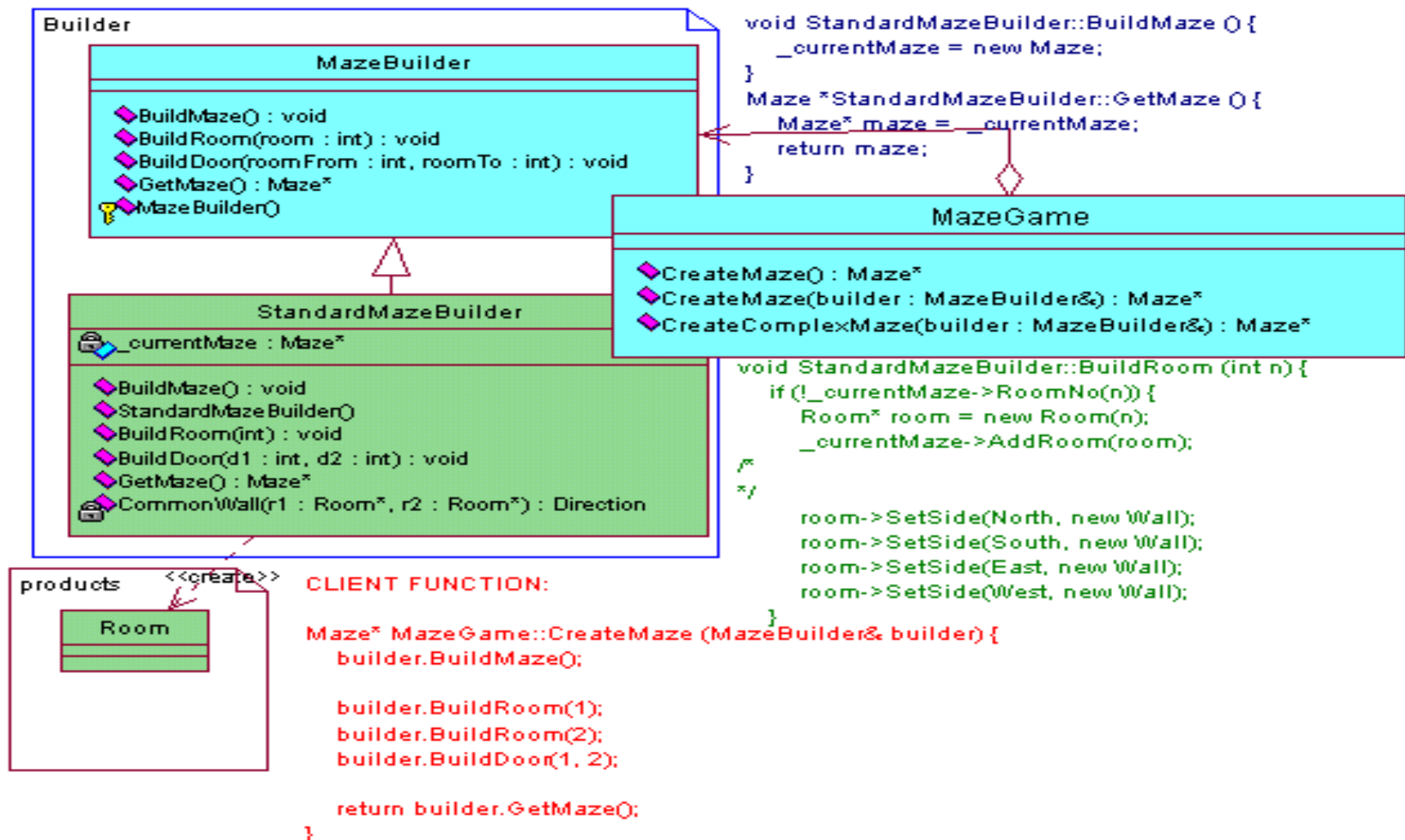
- ▶ The client creates the Director object and configures it with the desired Builder object
- ▶ Director notifies the builder whenever a part of the product should be built
- ▶ Builder handles requests for the director and adds parts to the products
- ▶ The client retrieves the product form the builder



# Consequences

- ▶ **It lets you vary a product's internal representation:**
  - director is provided with an abstract interface for constructing product.
  - Interfaces lets the builder hide the representation and internal structure and also how the product get assembled
- ▶ **It isolates code for contraction and representation:**
  - this pattern provide the modularity by encapsulating the way complex object is constructed and represented , client need not know anything about the classes that define the product's internal structure
  - ConcreteBuilder contains all the code to create and assemble a particular kind of product.
- ▶ **It gives you finer control over the construction process:**
  - builder constructs the product step by step under the director's control. Only when the product is finished does the director retrieve it from the builder.

# Model:





**Thank You!**