Software Design Patterns

Lecture 09

BIL428 Software Architectures

Mustafa Sert
Asst. Prof.
msert@baskent.edu.tr

Department of Computer Engineering, Başkent University
Ankara 06810 TURKEY

Outline

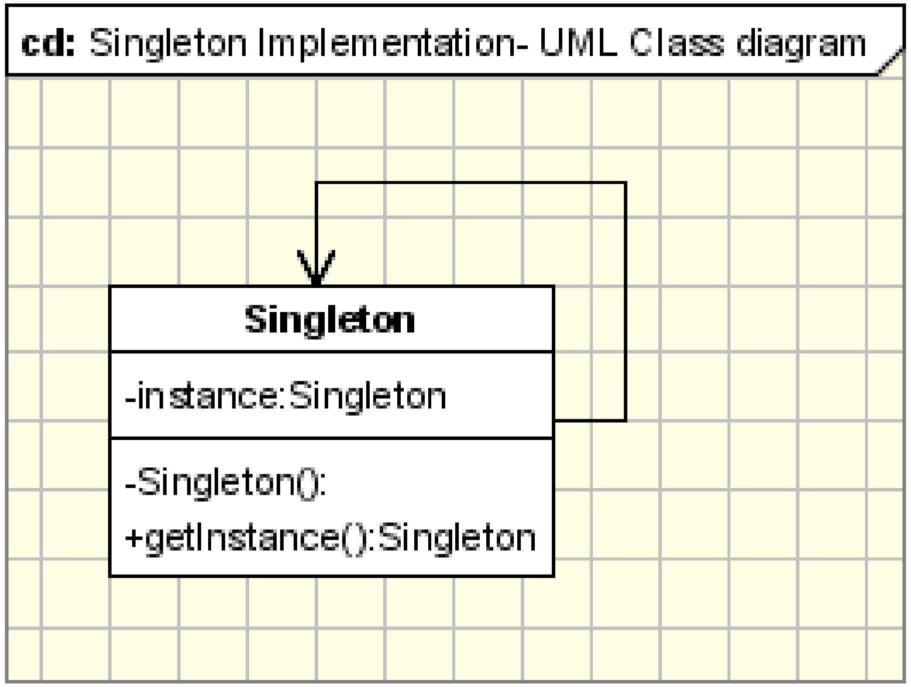
- Creational Patterns
  - Singleton
  - Factory
    - Factory Method
  - Prototype
  - Object Pool

Singleton

- A kind of **creational** design pattern
- It's important to have only one instance for a class. For example,
  - In a system there should be only one window manager
- Singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves
- Intent
  - Ensure that only one instance of a class is created
  - Provide a global point of access to the object

Singleton – Class Diagram

- The implementation involves
  - A static member in the "Singleton" class,
  - A private constructor
  - A static public method that returns a reference to the static member.



Singleton – Implementation

```
class Singleton
{
        private static Singleton instance;
        private Singleton()
        {
                ...
        }

        public static synchronized Singleton getInstance()
        {
                if (instance == null)
                        instance = new Singleton();

                return instance;
        }
        ...
        public void doSomething()
        {
                ...
        }
}
```

Singleton – Example Usage

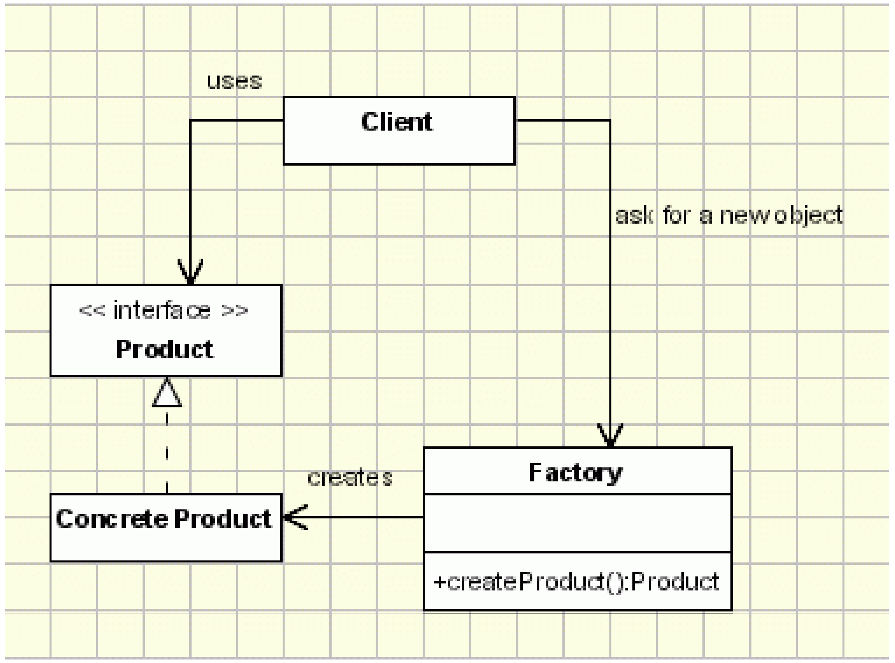
- **Logger Classes**
  - When need a global logging access point in all the application components without being necessary to create an object each time a logging operations is performed
- **Configuration Classes**
  - Provide a global access point, but also keep the instance in use as a cache object
  - When the class is instantiated (or when a value is read) the singleton will keep the values in its internal structure
  - If the values are read from the database or from files this avoids the reloading the values each time the configuration parameters are used

Factory Pattern

- A kind of **creational** design pattern
- It defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time
- **Intent**
  - Defines an interface for creating objects, but let subclasses to decide which class to instantiate
  - Refers to the newly created object through a common interface
- Two types of it
  - Factory Method
  - Abstract Factory

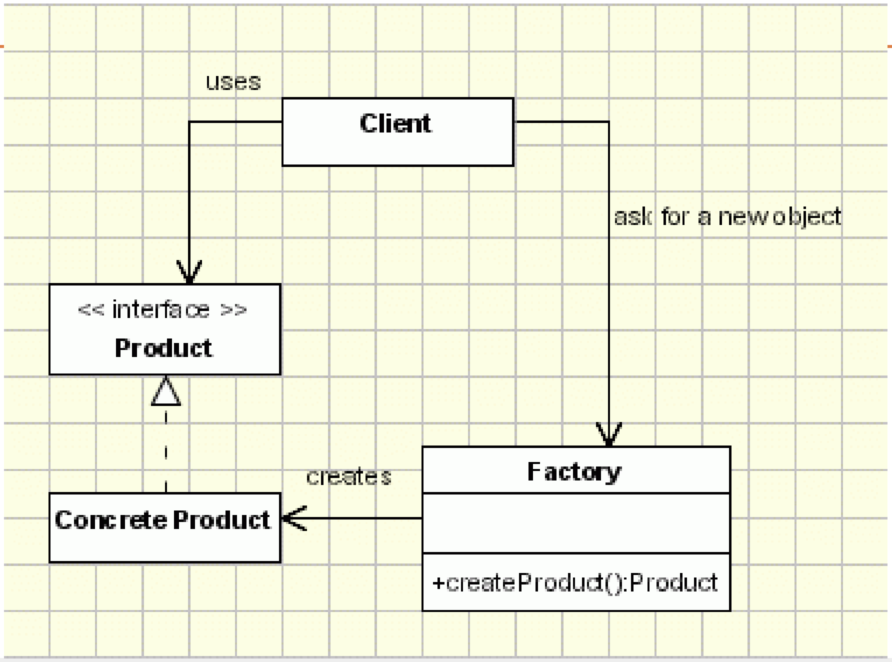
Factory Pattern – Basic Idea

- Implementation
  - The client needs a product, but instead of creating it directly using the new operator, it asks the factory object for a new product, providing the information about the type of object it needs.
  - The factory instantiates a new concrete product and then returns to the client the newly created product(casted to abstract product class).
  - The client uses the products as abstract products without being aware about their concrete implementation.



Factory Pattern – Basic Idea

- Procedural Solution
  - switch/case instantiation

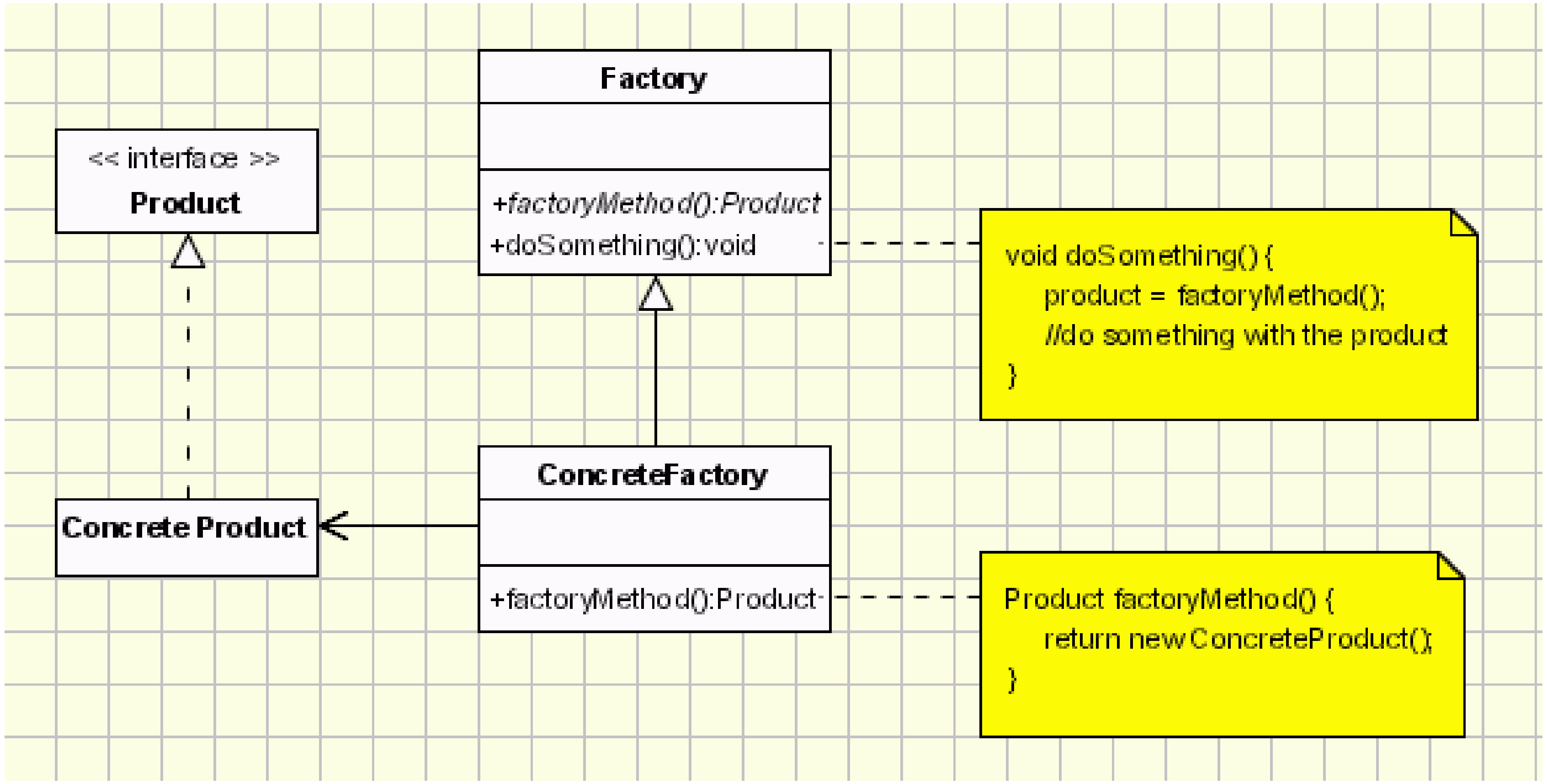


```
public class ProductFactory{
        public Product createProduct(String ProductID){
                if (id==ID1)
                        return new OneProduct();
                if (id==ID2) return
                        return new AnotherProduct();
                ... // so on for the other Ids

        return null; //if the id doesn't have any of the expected values
    }
    ...
}
```

Factory Pattern

- A more advanced solution - Factory design pattern with abstractions (**Factory Method**)
- For the **procedural switch-case implementation** we need to change the Factory class
- In **Factory Method** all we need is to register the class to the factory without actually modifying the factory class.

Factory Method – Class Diagram (1/2)

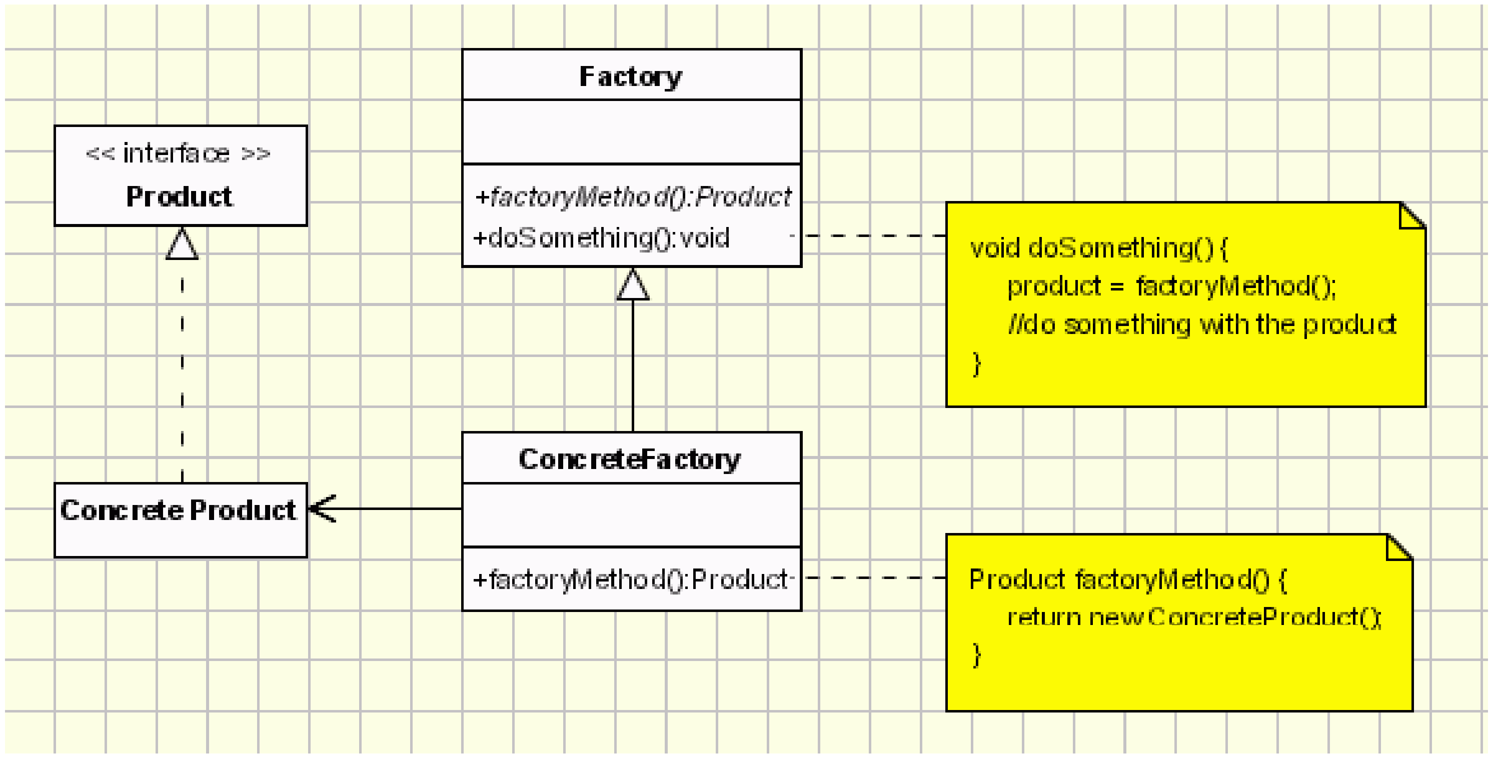


```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

- **Product** defines the interface for objects the factory method creates.
- **ConcreteProduct** implements the Product interface.

Factory Method – Class Diagram (2/2)



```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

- **Creator** (also refered as **Factory** because it creates the Product objects) declares the method **FactoryMethod**, which returns a Product object. May call the generating method for creating Product objects
- **ConcreteCreator** overrides the generating method for creating ConcreteProduct objects

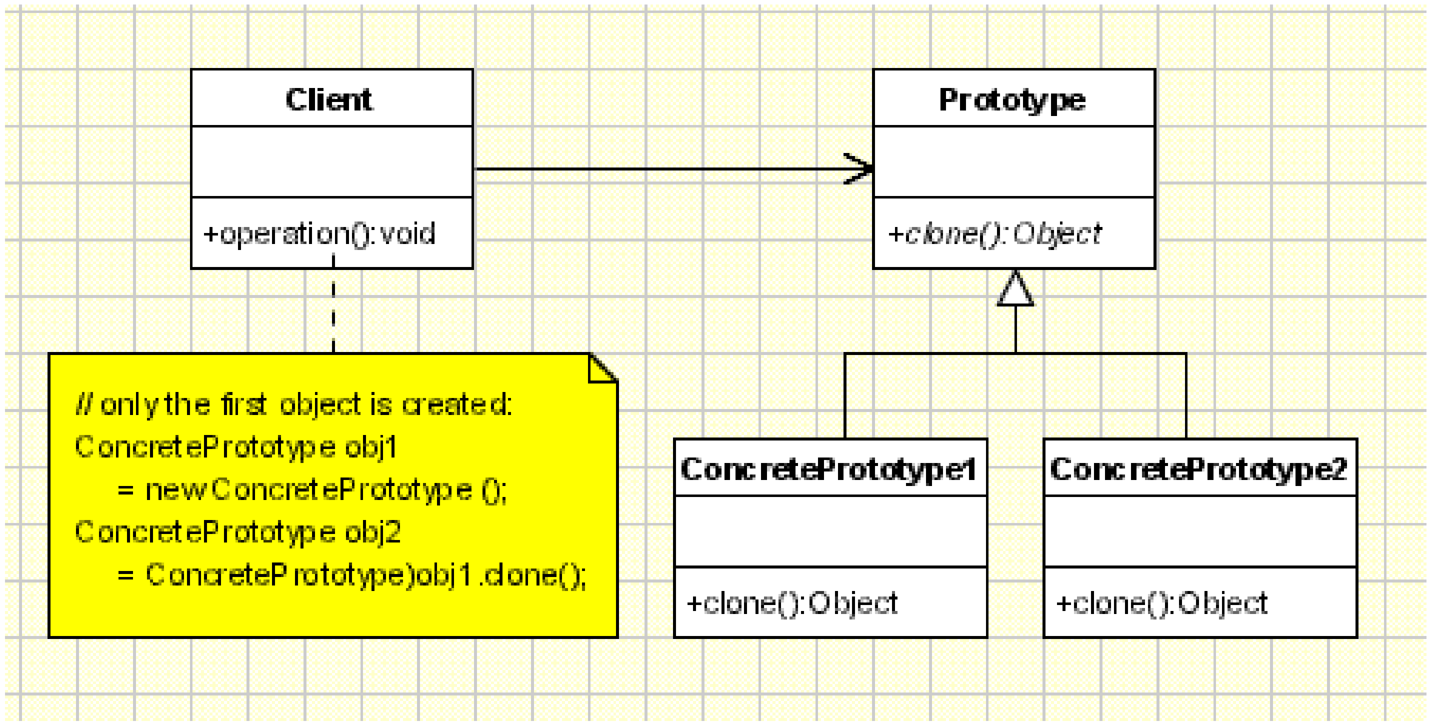
Factory Method – Implementation

```
public interface Product { � }

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

        protected abstract Product factoryMethod();
}

public class ConcreteProduct implements Product { � }

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();
        }
}
```

Prototype Pattern

- A kind of **creational** design pattern
- Prototype pattern helps in improving the performance by cloning the objects
- Intent
  - Specifying the kind of objects to create using a prototypical instance
  - creating new objects by copying this prototype

Prototype Pattern – Class Diagram



```
// only the first object is created:
ConcretePrototype obj1
    = new ConcretePrototype ();
ConcretePrototype obj2
    = ConcretePrototype)obj1.clone();
```

- 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.

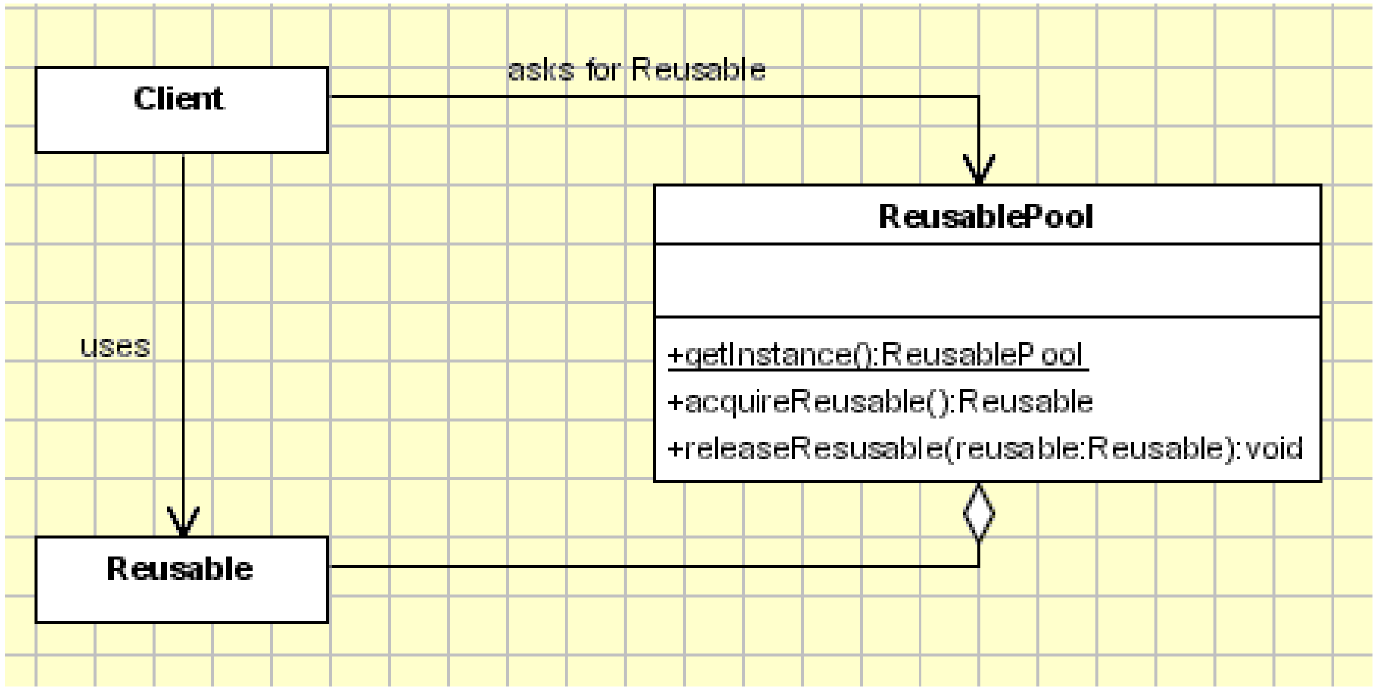
Prototype Pattern – Implementation

```
public interface Prototype {
    public abstract Object clone ( );
}

public class ConcretePrototype implements Prototype {
    public ConcretePrototype clone() {
        return (ConcretePrototype)super.clone();
    }
}

public class Client {

    public static void main( String arg[] )
    {
        ConcretePrototype obj1= new ConcretePrototype ();
        Prototype obj2 = (ConcretePrototype)obj1.clone();
    }
}
```

Object Pool

- A kind of **creational** design pattern
- Performance can be sometimes the key issue during the software development and the object creation (class instantiation) is a costly step.
- While the **Prototype** pattern helps in improving the performance by cloning the objects, the **Object Pool** pattern offer a mechanism to reuse objects that are expensive to create.
- Intent
  - Reuse and share objects that are expensive to create.

Object Pool – Class Diagram



asks for Reusable
Client
ReusablePool
+getInstance():ReusablePool
+acquireReusable():Reusable
+releaseResusable(reusable:Reusable):void
uses
Reusable

- **Implementation involves the following objects:**

  Reusable - Wraps the limited resource, will be shared by several clients for a limited amount of time.

  Client - uses an instance of type Reusable.

  ReusablePool - manage the reusable objects for use by Clients, creating and managing a pool of objects

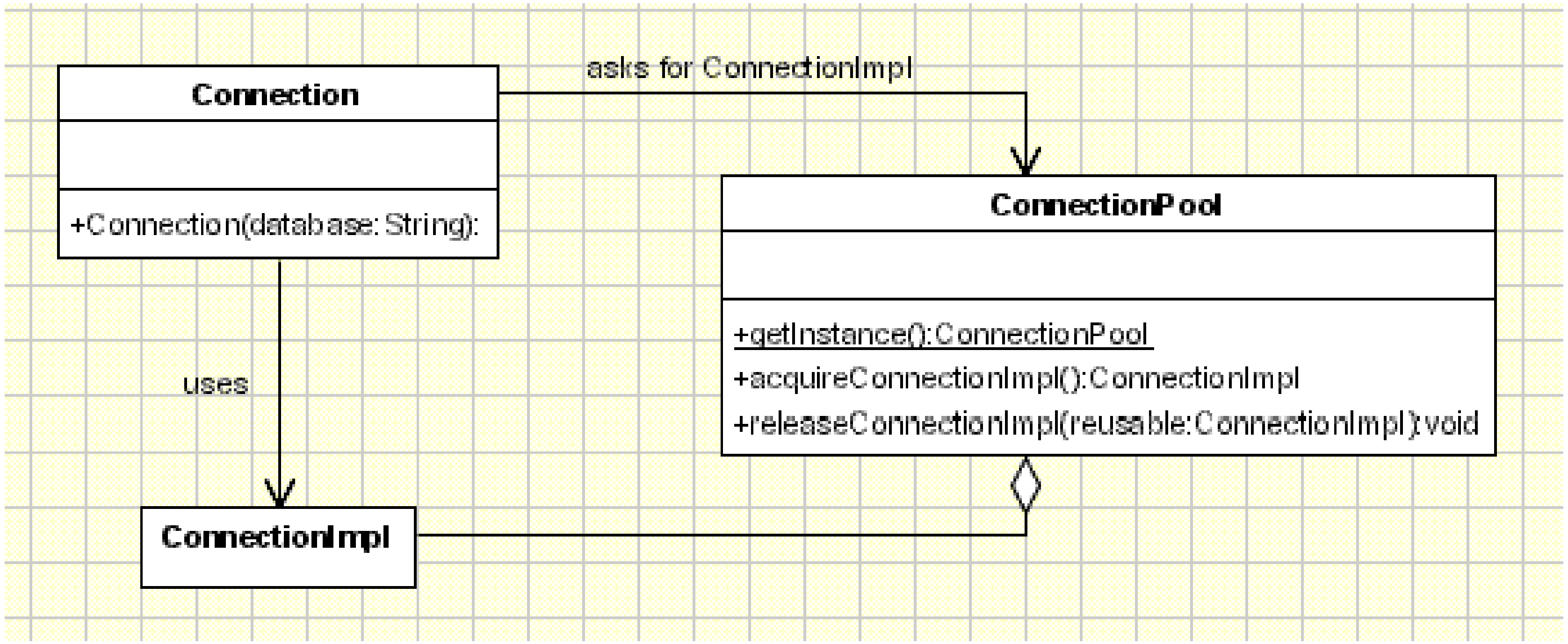
Object Pool – Implementation

- When a client asks for a Reusable object, the pool performs the following actions:
  - Search for an available Reusable object and if it was found it will be returned to the client.
  - If no Reusable object was found then it tries to create a new one. If this actions succeeds the new Reusable object will be returned to the client.
  - If the pool was unable to create a new Reusable, the pool will wait until a reusable object will be released.

Object Pool – Example

- Database connections
  - It's obviosly that opening too many connections might affect the performance for several reasons:
    - Creating a connection is an expensive operation.
    - When there are too many connections opened it takes longer to create a new one and the database server will become overloaded.

Object Pool – Example


Connection
+Connection(database: String):
asks for ConnectionImpl
ConnectionPool
+getInstance():ConnectionPool
+acquireConnectionImpl():ConnectionImpl
+releaseConnectionImpl(reusable:ConnectionImpl):void
uses
ConnectionImpl

- **Connection** - represent the object which is instantiated by the client. From the client perspective this object is instantiated and it handles the database operations and it is the only object visible to the client. The client is not aware that it uses some shared connections. Internally this class does not contain any code for connecting to the database and calls ConnectionPool.aquireImpl to get a ConnectionImpl object and then delegates the request to ConnectionImpl.

- ConnectionImpl is the object which implements the database operations which are exposed by Connection for the client.
- ConnectionPool is the main actor to manage the connections to the database. It keeps a list of ConnectionImpl objects and instantiates new objects if this is required.
- When the client needs to query the database it instantiate a new Connection object specifying the database name and the call the query method which returns a set of records. From the client point of view this is all.
  - When the Connection.Query method is called it asks for a ConnectionImpl object from the ConnectionPool. The ConnectionPool tries to find and return an unused object and if it doesn't find it creates one. At this point the maximum number of connections can be limited and if it was reached the pool cand wait until one will be available or return null. In the query method the request is delegated to the ConnectionImpl object returned by the object pool. Since the request is just delegated it's recomended to have the same method signature in Connection and ConnectionImpl.