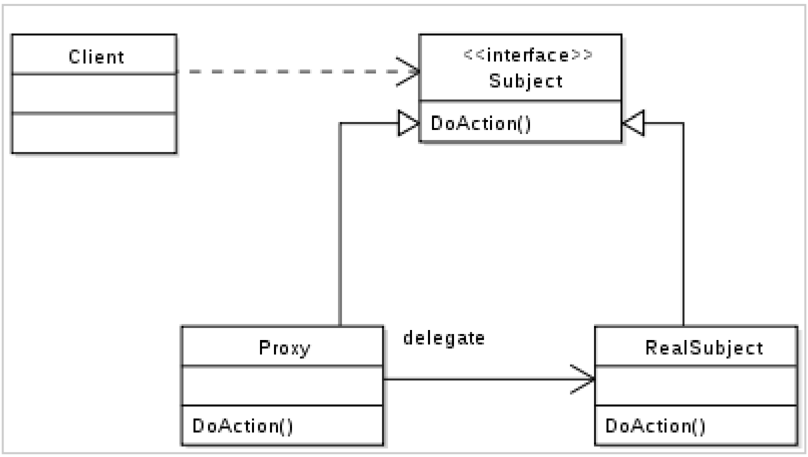
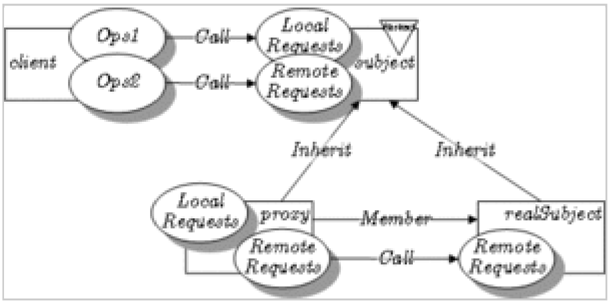
Proxy pattern

From Wikipedia, the free encyclopedia

In computer programming, the **proxy pattern** is a software design pattern.



Proxy in UML



Proxy in LePUS3 (legend (http://lepus.org.uk/ref/legend/legend.xml))

A *proxy*, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.

Contents

- 1 Possible Usage Scenarios
  - 1.1 Remote Proxy
  - 1.2 Virtual Proxy
  - 1.3 Protection Proxy
- 2 Example
  - 2.1 C#
  - 2.2 C++
  - 2.3 Crystal
  - 2.4 Delphi / Object Pascal
  - 2.5 Java
- 3 See also
- 4 References
- 5 External links

Possible Usage Scenarios

Remote Proxy

In distributed object communication, a local object represents a remote object (one that belongs to a different address space). The local object is a proxy for the remote object, and method invocation on the local object results in remote method invocation on the remote object. An example would be an ATM implementation, where the ATM might hold proxy objects for bank information that exists in the remote server.

Virtual Proxy

In place of a complex or heavy object, a skeleton representation may be advantageous in some cases. When an underlying image is huge in size, it may be represented using a virtual proxy object, loading the real object on demand.

Protection Proxy

A protection proxy might be used to control access to a resource based on access rights.

Example

C#

The Wikibook *Computer Science Design Patterns* has a page on the topic of: ***Proxy implementations in various languages***

```
interface ICar
{
    void DriveCar();
}

// Real Object
public class Car : ICar
{
    public void DriveCar()
    {
        Console.WriteLine("Car has been driven!");
    }
}

//Proxy Object
public class ProxyCar : ICar
{
    private Driver driver;
    private ICar realCar;

    public ProxyCar(Driver driver)
    {
        this.driver = driver;
        this.realCar = new Car();
    }

    public void DriveCar()
    {
        if (driver.Age <= 16)
            Console.WriteLine("Sorry, the driver is too young to drive.");
        else
            this.realCar.DriveCar();
     }
}

public class Driver
{
    private int Age { get; set; }

    public Driver(int age)
    {
        this.Age = age;
    }
}

// How to use above Proxy class?
private void btnProxy_Click(object sender, EventArgs e)
{
    ICar car = new ProxyCar(new Driver(16));
    car.DriveCar();

    car = new ProxyCar(new Driver(25));
    car.DriveCar();
}
```

Output

```
Sorry, the driver is too young to drive.
Car has been driven!
```

Notes:

- A proxy may hide information about the real object to the client.
- A proxy may perform optimization like on demand loading.
- A proxy may do additional house-keeping job like audit tasks.
- Proxy design pattern is also known as surrogate design pattern.

C++

```
class ICar {
public:
  virtual void DriveCar() = 0;
};

class Car : public ICar {
  void DriveCar() override {
    std::cout << "Car has been driven!" << std::endl;
  }
};

class ProxyCar : public ICar {
private:
  ICar* realCar;
  int _driver_age;

public:
  ProxyCar (int driver_age) : realCar(new Car()), _driver_age(driver_age) {}
  ~ProxyCar () {
    delete realCar;
  }

  void DriveCar() {
    if (_driver_age > 16)
      realCar->DriveCar();
    else
      std::cout << "Sorry, the driver is too young to drive." << std::endl;
  }
};

// How to use above Proxy class?
int main()
{
    ICar* car = new ProxyCar(16);
    car->DriveCar();
    delete car;

    car = new ProxyCar(25);
    car->DriveCar();
    delete car;
}
```

Crystal

```
abstract class AbstractCar
  abstract def drive
end

class Car < AbstractCar
  def drive
    puts "Car has been driven!"
  end
end

class Driver
  getter age : Int32

  def initialize(@age)
  end
end

class ProxyCar < AbstractCar
  private getter driver : Driver
  private getter real_car : AbstractCar

  def initialize(@driver)
    @real_car = Car.new
  end

  def drive
    if driver.age <= 16
      puts "Sorry, the driver is too young to drive."
    else
      @real_car.drive
    end
  end
end

# Program
driver = Driver.new(16)
car = ProxyCar.new(driver)
car.drive

driver = Driver.new(25)
car = ProxyCar.new(driver)
car.drive
```

Output

```
Sorry, the driver is too young to drive.
Car has been driven!
```

Delphi / Object Pascal

```
// Proxy Design pattern
unit DesignPattern.Proxy;

interface

type
    // Car Interface
    ICar = interface
      procedure DriveCar;
    end;

    // TCar class, implementing ICar
    TCar = Class(TInterfacedObject, ICar)
      class function New: ICar;
      procedure DriveCar;
    End;

    // Driver Interface
    IDriver = interface
      function Age: Integer;
    end;

    // TDriver Class, implementing IDriver
    TDriver = Class(TInterfacedObject, IDriver)
    private
      FAge: Integer;
    public
      constructor Create(Age: Integer); Overload;
      class function New(Age: Integer): IDriver;
      function Age: Integer;
    End;

    // Proxy Object
    TProxyCar = Class(TInterfacedObject, ICar)
    private
      FDriver: IDriver;
      FRealCar: ICar;
    public
```

```
      constructor Create(Driver: IDriver); Overload;
      class function New(Driver: IDriver): ICar;
      procedure DriveCar;
    End;

implementation

{ TCar Implementation }

class function TCar.New: ICar;
begin
     Result := Create;
end;

procedure TCar.DriveCar;
begin
     WriteLn('Car has been driven!');
end;

{ TDriver Implementation }

constructor TDriver.Create(Age: Integer);
begin
     inherited Create;
     FAge := Age;
end;

class function TDriver.New(Age: Integer): IDriver;
begin
     Result := Create(Age);
end;

function TDriver.Age: Integer;
begin
     Result := FAge;
end;

{ TProxyCar Implementation }

constructor TProxyCar.Create(Driver: IDriver);
begin
     inherited Create;
     Self.FDriver  := Driver;
     Self.FRealCar := TCar.Create AS ICar;
end;

class function TProxyCar.New(Driver: IDriver): ICar;
begin
     Result := Create(Driver);
end;

procedure TProxyCar.DriveCar;
begin
     if (FDriver.Age <= 16)
        then WriteLn('Sorry, the driver is too young to drive.')
        else FRealCar.DriveCar();
end;

end.
```

Usage

```
program Project1;
{$APPTYPE Console}
uses
    DesignPattern.Proxy in 'DesignPattern.Proxy.pas';
begin
     TProxyCar.New(TDriver.New(16)).DriveCar;
     TProxyCar.New(TDriver.New(25)).DriveCar;
end.
```

Output

```
Sorry, the driver is too young to drive.
Car has been driven!
```

Java

The following Java example illustrates the "virtual proxy" pattern. The `ProxyImage` class is used to access a remote method.

The example creates first an interface against which the pattern creates the classes. This interface contains only one method to display the image, called `displayImage()`, that has to be coded by all classes implementing it.

The proxy class `ProxyImage` is running on another system than the real image class itself and can represent the real image `RealImage` over there. The image information is accessed from the disk. Using the proxy pattern, the code of the `ProxyImage` avoids multiple loading of the image, accessing it from the other system in a memory-saving manner. It should be noted, however, that the lazy loading demonstrated in this example is not part of the proxy pattern, but is merely an advantage made possible by the use of the proxy.

```
interface Image {
    public void displayImage();
}

// On System A
class RealImage implements Image {

    private String filename = null;
    /**
     * Constructor
     * @param filename
     */
    public RealImage(final String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    /**
     * Loads the image from the disk
     */
    private void loadImageFromDisk() {
        System.out.println("Loading   " + filename);
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        System.out.println("Displaying " + filename);
    }

}

// On System B
class ProxyImage implements Image {

    private RealImage image = null;
    private String filename = null;
    /**
     * Constructor
     * @param filename
     */
    public ProxyImage(final String filename) {
        this.filename = filename;
    }

    /**
     * Displays the image
     */
    public void displayImage() {
        if (image == null) {
           image = new RealImage(filename);
        }
        image.displayImage();
    }

}

class ProxyExample {

   /**
    * Test method
    */
   public static void main(final String[] arguments) {
        final Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        final Image image2 = new ProxyImage("HiRes_10MB_Photo2");

        image1.displayImage(); // loading necessary
        image1.displayImage(); // loading unnecessary
        image2.displayImage(); // loading necessary
        image2.displayImage(); // loading unnecessary
        image1.displayImage(); // loading unnecessary
    }
}
```

The program's output is:

```
Loading   HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Displaying HiRes_10MB_Photo1
Loading   HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo2
Displaying HiRes_10MB_Photo1
```

See also

- Composite pattern
- Decorator pattern
- Lazy initialization

References

External links

- Proxy pattern in UML and in LePUS3 (a formal modelling language) (http://www.lepus.org.uk/ref/companion/Proxy.xml)
- Take control with the Proxy design pattern (http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html) by David Geary, JavaWorld.com
- PerfectJPattern Open Source Project (http://perfectjpattern.sourceforge.net/dp-proxy.html), Provides componentized implementation of the Proxy Pattern in Java
- Adapter vs. Proxy vs. Facade Pattern Comparison (http://www.netobjectives.com/PatternRepository/index.php?title=AdapterVersusProxyVersusFacadePatternComparison)
- Proxy Design Pattern (http://sourcemaking.com/design_patterns/proxy)
- Proxy pattern C++ implementation example (http://www.patterns.org.pl/proxy.html)
- Proxy pattern description from the Portland Pattern Repository (http://c2.com/cgi/wiki?ProxyPattern)

Wikimedia Commons has media related to ***Proxy pattern***.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Proxy_pattern&oldid=783573577"

Categories: Software design patterns