Creational patterns

These patterns have to do with class instantiation. They can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation to get the job done.

Abstract Factory groups object factories that have a common theme.
Builder constructs complex objects by separating construction and representation.
Factory Method creates objects without specifying the exact class to create.
Prototype creates objects by cloning an existing object.
Singleton restricts object creation for a class to only one instance.

Structural patterns

These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
Bridge decouples an abstraction from its implementation so that the two can vary independently.
Composite composes one-or-more similar objects so that they can be manipulated as one object.
Decorator dynamically adds/overrides behaviour in an existing method of an object.
Facade provides a simplified interface to a large body of code.
Flyweight reduces the cost of creating and manipulating a large number of similar objects.
Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Behavioral patterns

These design patterns are about classes objects communication. They are specifically concerned with communication between **objects**.

Chain of responsibility delegates commands to a chain of processing objects.
Command creates objects which encapsulate actions and parameters.
Interpreter implements a specialized language.
Iterator accesses the elements of an object sequentially without exposing its underlying representation.
Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
Memento provides the ability to restore an object to its previous state (undo).

Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.
State allows an object to alter its behavior when its internal state changes.
Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Abstract factory pattern

From Wikipedia, the free encyclopedia

Jump to: navigation, search

A software design pattern, the **Abstract Factory Pattern** provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software would create a concrete implementation of the abstract factory and then use the generic interfaces to create the concrete objects that are part of the theme. The client does not know (or care) about which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from its general usage.

An example of this would be an abstract factory class *DocumentCreator* that provides interfaces to create a number of products (eg. *createLetter()* and *createResume()*). The system would have any number of derived concrete versions of the *DocumentCreator* class like *FancyDocumentCreator* or *ModernDocumentCreator*, each with a different implementation of *createLetter()* and *createResume()* that would create a corresponding object like *FancyLetter* or *ModernResume*. Each of these products is derived from a simple abstract class like *Letter* or *Resume* of which the client is aware. The client code would get an appropriate instantiation of the *DocumentCreator* and call its factory methods. Each of the resulting objects would be created from the same DocumentCreator implementation and would share a common theme (they would all be fancy or modern objects). The client would need to know how to handle only the abstract *Letter* or *Resume* class, not the specific version that it got from the concrete factory.

In software development, a **Factory** is the location in the code at which objects are constructed. The intent in employing the pattern is to insulate the creation of objects from their usage. This allows for new derived types to be introduced with no change to the code that uses the base class.

Use of this pattern makes it possible to interchange concrete classes without changing the code that uses them, even at runtime. However, employment of this pattern, as with similar design patterns, may result in unnecessary complexity and extra work in the initial writing of code.

How to use it

The *factory* determines the actual *concrete* type of object to be created, and it is here that the object is actually created (in C++, for instance, by the **new** operator). However, the factory only returns an *abstract* pointer to the created concrete object.

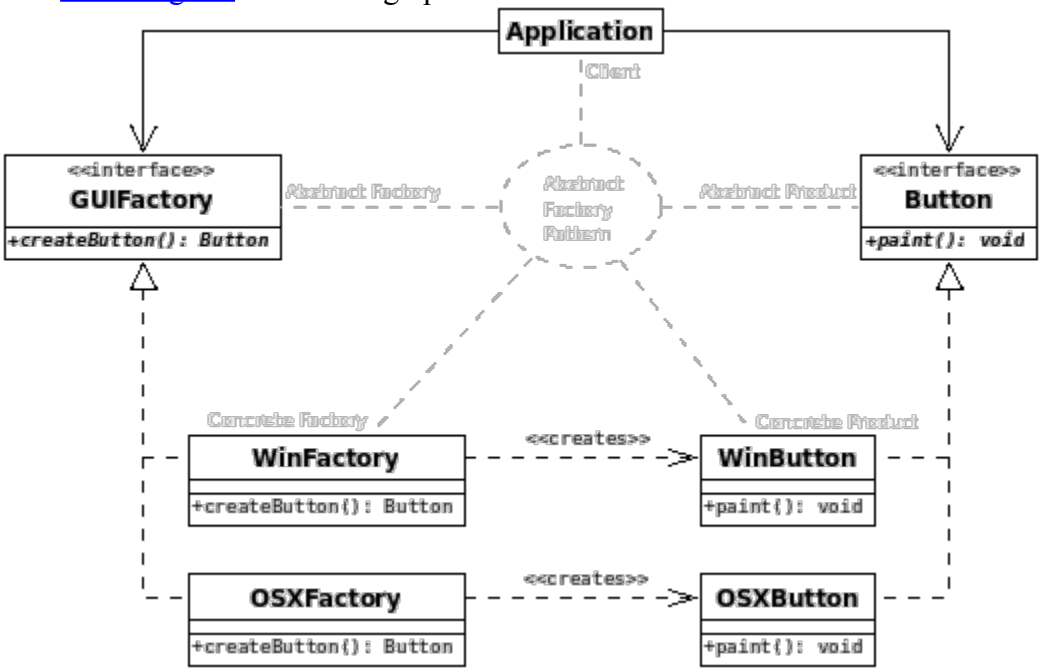
This insulates client code from object creation by having clients ask a factory object to create an object of the desired abstract type and to return an abstract pointer to the object.

As the factory only returns an abstract pointer, the client code (which requested the object from the factory) does not know - and is not burdened by - the actual concrete type of the object which was just created. However, the type of a concrete object (and hence a concrete factory) is known by the abstract factory; for instance, the factory may read it from a configuration file. The client has no need to specify the type, since it has already been specified in the configuration file. In particular, this means:

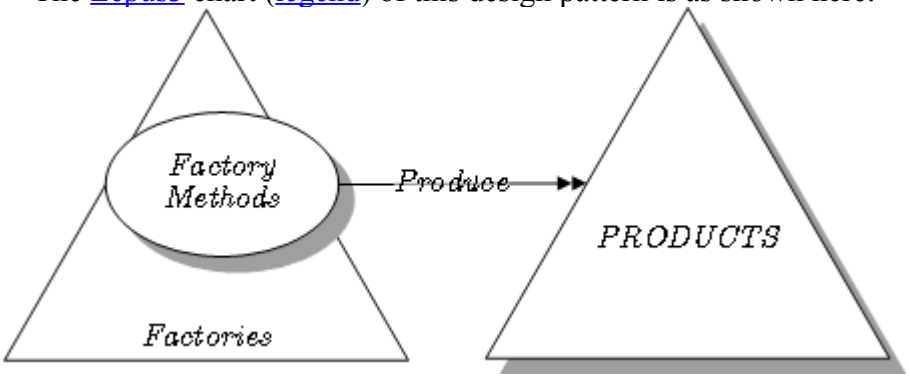
> The client code has no knowledge whatsoever of the concrete type, not needing to include any header files or class declarations relating to the concrete type. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interface.
> Adding new concrete types is done by modifying the client code to use a different factory, a modification which is typically one line in one file. (The different factory then creates objects of a *different* concrete type, but still returns a pointer of the *same* abstract type as before - thus insulating the client code from change.) This is significantly easier than modifying the client code to instantiate a new type, which would require changing *every* location in the code where a new object is created (as well as making sure that all such code locations also have knowledge of the new concrete type, by including for instance a concrete class header file). If all factory objects are stored globally in a singleton object, and all client code goes through the singleton to access the proper factory for object creation, then changing factories is as easy as changing the singleton object.

Structure

The class diagram of this design pattern is as shown here:



The Lepus3 chart (legend) of this design pattern is as shown here:



This class diagram does not emphasize the usage of abstract factory pattern in creating families of related or non related objects.

Example

Java

```
/*
* GUIFactory example
*/
abstract class GUIFactory {
public static GUIFactory getFactory() {
int sys = readFromConfigFile("OS_TYPE");
if (sys == 0) {
return new WinFactory();
} else {
return new OSXFactory();
```

```
                    }
                    }

    public abstract Button createButton();
                    }


    class WinFactory extends GUIFactory {
        public Button createButton() {
            return new WinButton();
                    }
                    }


    class OSXFactory extends GUIFactory {
        public Button createButton() {
            return new OSXButton();
                    }
                    }



            abstract class Button {
        public abstract void paint();
                    }


        class WinButton extends Button {
              public void paint() {
    System.out.println("I'm a WinButton");
                    }
                    }


        class OSXButton extends Button {
              public void paint() {
    System.out.println("I'm an OSXButton");
                    }
                    }


            public class Application {
      public static void main(String[] args) {
GUIFactory factory = GUIFactory.getFactory();
      Button button = factory.createButton();
                  button.paint();
                    }
               // Output is either:
              //    "I'm a WinButton"
                     // or:
             //    "I'm an OSXButton"
                    }
```

Builder pattern

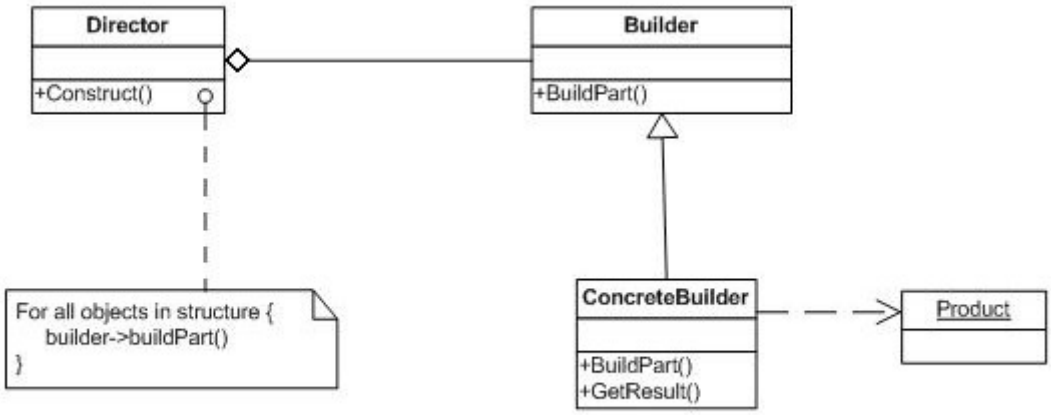
From Wikipedia, the free encyclopedia

Jump to: navigation, search

The **Builder Pattern** is a software design pattern. The intention is to separate the construction of a complex object from its representation so that the same construction process can create different representations.

Often, the **Builder Pattern** is used to build Products in accordance to the Composite pattern, a structure pattern.

Class Diagram



Builder

Abstract interface for creating objects(product).

Concrete Builder

Provide implementation for Builder. Construct and assemble parts to build the objects.

Director

The Director class is responsible for managing the correct sequence of object creation. It receives a Concrete Builder as a parameter and executes the necessary operations on it.

Product

The complex object under construction.

Useful tips

> Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
> Builder often builds a Composite.
> Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory,

Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

Examples

Java

```
/** "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;
        }
}

/** "Abstract 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 Cook {
        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 given type of pizza being constructed. */
public class BuilderExample {
        public static void main(String[] args) {
                Cook cook = new Cook();
                PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
                PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

                cook.setPizzaBuilder(hawaiianPizzaBuilder);
                cook.constructPizza();

                Pizza pizza = cook.getPizza();
        }
}
```

C#

```
//Implementation in C#.
class Pizza
{
    string dough;
    string sauce;
    string topping;
    public Pizza() {}
    public void SetDough( string d){ dough = d;}
    public void SetSauce( string s){ sauce = s;}
    public void SetTopping( string t){ topping = t;}
}

//Abstract Builder
abstract class PizzaBuilder
{
    protected Pizza pizza;
    public PizzaBuilder(){}
    public Pizza GetPizza(){ return pizza; }
    public void CreateNewPizza() { pizza = new Pizza(); }

    public abstract void BuildDough();
    public abstract void BuildSauce();
    public abstract void BuildTopping();
```

```
}

//Concrete Builder
class HawaiianPizzaBuilder : PizzaBuilder
{
    public override void BuildDough()   { pizza.SetDough("cross"); }
    public override void BuildSauce()   { pizza.SetSauce("mild"); }
    public override void BuildTopping() { pizza.SetTopping("ham+pineapple"); }
}

//Concrete Builder
class SpicyPizzaBuilder : PizzaBuilder
{
    public override void BuildDough()   { pizza.SetDough("pan baked"); }
    public override void BuildSauce()   { pizza.SetSauce("hot"); }
    public override void BuildTopping() { pizza.SetTopping("pepparoni+salami"); }
}

/** "Director" */
class Waiter {
    private PizzaBuilder pizzaBuilder;

    public void SetPizzaBuilder (PizzaBuilder pb) { pizzaBuilder = pb; }
    public Pizza GetPizza() { return pizzaBuilder.GetPizza(); }

    public void ConstructPizza() {
        pizzaBuilder.CreateNewPizza();
        pizzaBuilder.BuildDough();
        pizzaBuilder.BuildSauce();
        pizzaBuilder.BuildTopping();
    }
}

/** A customer ordering a pizza. */
class BuilderExample
{
    public static void Main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        waiter.SetPizzaBuilder ( hawaiianPizzaBuilder );
        waiter.ConstructPizza();

        Pizza pizza = waiter.GetPizza();
    }
}
```

C++

```
// Implementation in C++.

#include <iostream>
#include <memory>
#include <string>

// Product
class Pizza
{
private:
    std::string dough;
    std::string sauce;
    std::string topping;

public:
    Pizza() { }
    ~Pizza() { }

    void SetDough(const std::string& d) { dough = d; };
    void SetSauce(const std::string& s) { sauce = s; };
    void SetTopping(const std::string& t) { topping = t; }
```

```
    void ShowPizza()
    {
        std::cout << " Yummy !!!" << std::endl
        << "Pizza with Dough as " << dough
        << ", Sauce as " << sauce
        << " and Topping as " << topping
        << " !!! " << std::endl;
    }
};

// Abstract Builder
class PizzaBuilder
{
protected:
    std::auto_ptr<Pizza> pizza;
public:
    PizzaBuilder() {}
    virtual ~PizzaBuilder() {}
    std::auto_ptr<Pizza> GetPizza() { return pizza; }

    void createNewPizzaProduct() { pizza.reset (new Pizza); }

    virtual void buildDough()=0;
    virtual void buildSauce()=0;
    virtual void buildTopping()=0;
};

// ConcreteBuilder
class HawaiianPizzaBuilder : public PizzaBuilder
{
public:
    HawaiianPizzaBuilder() : PizzaBuilder() {}
    ~HawaiianPizzaBuilder(){}

    void buildDough() { pizza->SetDough("cross"); }
    void buildSauce() { pizza->SetSauce("mild"); }
    void buildTopping() { pizza->SetTopping("ham and pineapple"); }
};

// ConcreteBuilder
class SpicyPizzaBuilder : public PizzaBuilder
{
public:
    SpicyPizzaBuilder() : PizzaBuilder() {}
    ~SpicyPizzaBuilder() {}

    void buildDough() { pizza->SetDough("pan baked"); }
    void buildSauce() { pizza->SetSauce("hot"); }
    void buildTopping() { pizza->SetTopping("pepperoni and salami"); }
};

// Director
class Waiter
{
private:
    PizzaBuilder* pizzaBuilder;
public:
    Waiter() : pizzaBuilder(NULL) {}
    ~Waiter() { }

    void SetPizzaBuilder(PizzaBuilder* b) { pizzaBuilder = b; }
    std::auto_ptr<Pizza> GetPizza() { return pizzaBuilder->GetPizza(); }
    void ConstructPizza()
    {
        pizzaBuilder->createNewPizzaProduct();
        pizzaBuilder->buildDough();
        pizzaBuilder->buildSauce();
        pizzaBuilder->buildTopping();
    }
};
```

```
// A customer ordering a pizza.
int main()
{
    Waiter waiter;

    HawaiianPizzaBuilder hawaiianPizzaBuilder;
    waiter.SetPizzaBuilder (&hawaiianPizzaBuilder);
    waiter.ConstructPizza();
    std::auto_ptr<Pizza> pizza = waiter.GetPizza();
    pizza->ShowPizza();

    SpicyPizzaBuilder spicyPizzaBuilder;
    waiter.SetPizzaBuilder(&spicyPizzaBuilder);
    waiter.ConstructPizza();
    pizza = waiter.GetPizza();
    pizza->ShowPizza();

    return EXIT_SUCCESS;
}
```

Visual Prolog

Product

```
interface pizza
   predicates
      setDough : (string Dough).
      setSauce : (string Sauce).
      setTopping : (string Topping).
end interface pizza

class pizza : pizza
end class pizza

implement pizza
   facts
      dough : string := "".
      sauce : string := "".
      topping : string := "".
   clauses
      setDough(Dough) :- dough := Dough.
   clauses
      setSauce(Sauce) :- sauce := Sauce.
   clauses
      setTopping(Topping) :- topping := Topping.
end implement pizza
```

Abstract Builder

```
interface pizzaBuilder
   predicates
      getPizza : () -> pizza Pizza.
      createNewPizzaProduct : ().
   predicates
      buildDough : ().
      buildSauce : ().
      buildTopping : ().
end interface pizzaBuilder
```

Visual Prolog does not support abstract classes, but we can create a support class instead:

```
interface pizzaBuilderSupport
```

```
   predicates from pizzaBuilder
      getPizza, createNewPizzaProduct
end interface pizzaBuilderSupport

class pizzaBuilderSupport : pizzaBuilderSupport
end class pizzaBuilderSupport

implement pizzaBuilderSupport
   facts
      pizza : pizza := erroneous.
   clauses
      getPizza() = pizza.
   clauses
      createNewPizzaProduct() :- pizza := pizza::new().
end implement pizzaBuilderSupport
```

ConcreteBuilder #1

```
class hawaiianPizzaBuilder :  pizzaBuilder
end class hawaiianPizzaBuilder

implement hawaiianPizzaBuilder
   inherits pizzaBuilderSupport

   clauses
      buildDough() :- getPizza():setDough("cross").
   clauses
      buildSauce() :- getPizza():setSauce("mild").
   clauses
      buildTopping() :- getPizza():setTopping("ham+pineapple").
end implement hawaiianPizzaBuilder
```

ConcreteBuilder #2

```
class spicyPizzaBuilder :  pizzaBuilder
end class spicyPizzaBuilder

implement spicyPizzaBuilder
   inherits pizzaBuilderSupport

   clauses
      buildDough() :- getPizza():setDough("pan baked").
   clauses
      buildSauce() :- getPizza():setSauce("hot").
   clauses
      buildTopping() :- getPizza():setTopping("pepperoni+salami").
end implement spicyPizzaBuilder
```

Director

```
interface waiter
   predicates
      setPizzaBuilder : (pizzaBuilder PizzaBuilder).
      getPizza : () -> pizza Pizza.
   predicates
      constructPizza : ().
end interface waiter

class waiter : waiter
end class waiter

implement waiter
   facts
```

```
      pizzaBuilder : pizzaBuilder := erroneous.
   clauses
      setPizzaBuilder(PizzaBuilder) :- pizzaBuilder := PizzaBuilder.
   clauses
      getPizza() = pizzaBuilder:getPizza().
   clauses
      constructPizza() :-
         pizzaBuilder:createNewPizzaProduct(),
         pizzaBuilder:buildDough(),
         pizzaBuilder:buildSauce(),
         pizzaBuilder:buildTopping().
end implement waiter
```

A customer ordering a pizza.

```
goal
   Hawaiian_pizzabuilder = hawaiianPizzaBuilder::new(),
   Waiter = waiter::new(),
   Waiter:setPizzaBuilder(Hawaiian_pizzabuilder),
   Waiter:constructPizza(),
   Pizza = Waiter:getPizza().
```

perl

```
## Product
package pizza;

sub new {
    return bless {
        dough => undef,
        sauce => undef,
        topping => undef
    }, shift;
}

sub set_dough {
    my( $self, $dough ) = @_;
    $self->{dough} = $dough;
}

sub set_sauce {
    my( $self, $sauce ) = @_;
    $self->{sauce} = $sauce;
}

sub set_topping {
    my( $self, $topping ) = @_;
    $self->{topping} = $topping;
}

1;


## Abstract builder
package pizza_builder;

sub new {
    return bless {
        pizza => undef
    }, shift;
}

sub get_pizza {
    my( $self ) = @_;
    return $self->{pizza};
}

sub create_new_pizza_product {
```

```
    my( $self ) = @_;
    $self->{pizza} = pizza->new;
}

# This is what an abstract method could look like in perl...

sub build_dough {
    croak("This method must be overridden.");
}

sub build_sauce {
    croak("This method must be overridden.");
}

sub build_topping {
    croak("This method must be overridden.");
}

1;


## Concrete builder
package hawaiian_pizza_builder;

use base qw{ pizza_builder };

sub build_dough {
    my( $self ) = @_;
    $self->{pizza}->set_dough("cross");
}

sub build_sauce {
    my( $self ) = @_;
    $self->{pizza}->set_sauce("mild");
}

sub build_topping {
    my( $self ) = @_;
    $self->{pizza}->set_topping("ham+pineapple");
}

1;


## Concrete builder
package spicy_pizza_builder;

use base qw{ pizza_builder };

sub build_dough {
    my( $self ) = @_;
    $self->{pizza}->set_dough("pan baked");
}

sub build_sauce {
    my( $self ) = @_;
    $self->{pizza}->set_sauce("hot");
}

sub build_topping {
    my( $self ) = @_;
    $self->{pizza}->set_topping("pepperoni+salami");
}

1;


## Director
package waiter;

sub new {
```

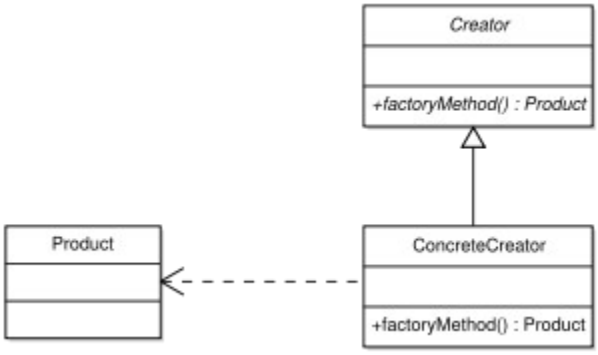
```
    return bless {
        pizza_builder => undef
    }, shift;
}

sub set_pizza_builder {
    my( $self, $builder ) = @_;
    $self->{pizza_builder} = $builder;
}

sub get_pizza {
    my( $self ) = @_;
    return $self->{pizza_builder}->get_pizza;
}

sub construct_pizza {
    my( $self ) = @_;
    $self->{pizza_builder}->create_new_pizza_product;
    $self->{pizza_builder}->build_dough;
    $self->{pizza_builder}->build_sauce;
    $self->{pizza_builder}->build_topping;
}

1;


## Lets order pizza (client of Director/Builder)
package main

my $waiter = waiter->new;
my $hawaiian_pb = hawaiian_pizza_builder->new;
my $spicy_pb = spicy_pizza_builder->new;

$waiter->set_pizza_builder( $hawaiian_pb );
$waiter->construct_pizza;

my $pizza = $waiter->get_pizza;

print "Serving a nice pizza with:\n";
for (keys %$pizza) {
    print "  $pizza->{$_} $_\n";
}

1;
```

PHP

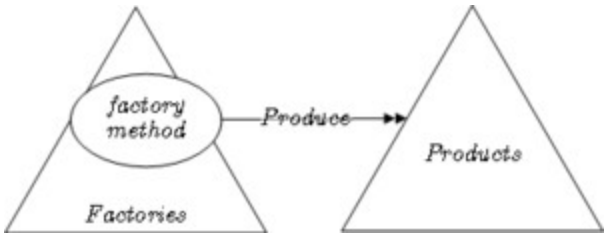
```
/** Product **/
class Pizza{
  private $dough;
  private $sauce;
  private $topping;
  public function setDough($dough){
    $this->dough = $dough;
  }
  public function setSauce($sauce){
    $this->sauce = $sauce;
  }
  public function setTopping($topping){
    $this->topping = $topping;
  }
}

/** Abstract builder **/
abstract class PizzaBuilder{
  protected $pizza;
  public function __construct(){
```

```
    $this->pizza = new Pizza();
  }
  public function getPizza(){
    return $this->pizza;
  }
  abstract function buildDough();
  abstract function buildSauce();
  abstract function buildTopping();
}

/** Concrete builder  **/
class SpicyPizza extends PizzaBuilder{
  public function buildDough(){
    $this->pizza->setDough('crispy');
  }
  public function buildSauce(){
    $this->pizza->setSauce('hot');
  }
  public function buildTopping(){
    $this->pizza->setTopping('pepperoni+salami');
  }
}

/** Director **/
class Chef{
  private $pizza_builder;
  public function setPizzaBuilder(PizzaBuilder $pizza_builder){
    $this->pizza_builder = $pizza_builder;
  }
  public function cookPizza(){
    $this->pizza_builder->buildDough();
    $this->pizza_builder->buildSauce();
    $this->pizza_builder->buildTopping();
  }
  public function getPizza(){
    return $this->pizza_builder->getPizza();
  }
}

//Customer orders a Pizza.
$chef = new Chef();

$order = new SpicyPizza();
$chef->setPizzaBuilder($order);
$chef->cookPizza();
$pizza = $chef->getPizza();

print_r($pizza);
```

Factory method pattern



Factory method in UML



Factory Method in LePUS3

The **factory method pattern** is an object-oriented design pattern. Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The factory method design pattern handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created. More generally, the term *factory method* is often used to refer to any method whose main purpose is creation of objects.

Definition

The essence of the Factory Pattern is to "Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses."

Common usage

> Factory methods are common in **toolkits** and **frameworks** where library code needs to create objects of types which may be subclassed by applications using the framework.
> Parallel class hierarchies often require objects from one hierarchy to be able to create appropriate objects from another.

Other benefits and variants

Although the motivation behind the factory method pattern is to allow subclasses to choose which type of object to create, there are other benefits to using factory methods, many of which do not depend on subclassing. Therefore, it is common to define "factory methods" that are not polymorphic to create objects in order to gain these other benefits. Such methods are often static.

Encapsulation

Factory methods encapsulate the creation of objects. This can be useful if the creation process is very complex, for example if it depends on settings in configuration files or on user input.

Consider as an example a program to read image files and make thumbnails out of them. The program supports different image formats, represented by a reader class for each format:

```
public interface ImageReader
{
     public DecodedImage getDecodedImage();
}

public class GifReader implements ImageReader
{
     private GifReader( InputStream in )
     {
         // check that it's a gif, throw exception if it's not, then if it is
         // decode it.
     }

     public DecodedImage getDecodedImage()
     {
        return decodedImage;
     }
}

public class JpegReader implements ImageReader
{
     //....
}
```

Each time the program reads an image it needs to create a reader of the appropriate type based on some information in the file. This logic can be encapsulated in a factory method:

```
public class ImageReaderFactory
{
    public static ImageReader getImageReader( InputStream is )
    {
        int imageType = figureOutImageType( is );

        switch( imageType )
        {
            case ImageReaderFactory.GIF:
                return new GifReader( is );
            case ImageReaderFactory.JPEG:
                return new JpegReader( is );
            // etc.
        }
    }
}
```

The code fragment in the previous example uses a switch statement to associate an `imageType` with a specific factory object. Alternatively, this association could also be implemented as a mapping. This would allow the switch statement to be replaced with an associative array lookup.

Limitations

There are three limitations associated with the use of the factory method. The first relates to refactoring existing code; the other two relate to inheritance.

The first limitation is that refactoring an existing class to use factories breaks existing clients. For example, if class Complex was a standard class, it might have numerous clients with code like:

```
Complex c = new Complex(-1, 0);
```

Once we realize that two different factories are needed, we change the class (to the code shown earlier). But since the constructor is now private, the existing client code no longer compiles.

The second limitation is that, since the pattern relies on using a private constructor, the class cannot be extended. Any subclass must invoke the inherited constructor, but this cannot be done if that constructor is private.

The third limitation is that, if we do extend the class (e.g., by making the constructor protected -- this is risky but possible), the subclass must provide its own re-implementation of all factory methods with exactly the same signatures. For example, if class StrangeComplex extends Complex, then unless StrangeComplex provides its own version of all factory methods, the call StrangeComplex.fromPolar(1, pi) will yield an instance of *Complex* (the superclass) rather than the expected instance of the subclass.

All three problems could be alleviated by altering the underlying programming language to make factories first-class class members (rather than using the design pattern).

Examples

C#

Pizza example:

```
public abstract class Pizza
{
    public abstract decimal GetPrice();

    public enum Types
    {
        HamMushroom, Deluxe, Seafood
    }
    public static Pizza PizzaFactory(Types t)
    {
        switch (t)
        {
            case Types.HamMushroom:
                return new HamAndMushroomPizza();

            case Types.Deluxe:
                return new DeluxePizza();

            case Types.Seafood:
                return new SeafoodPizza();

        }

        throw new System.NotSupportedException("The pizza type " + t.ToString() + " is
not recognized.");
```

```
    }
}
public class HamAndMushroomPizza : Pizza
{
    private decimal m_Price = 8.5M;
    public override decimal GetPrice() { return m_Price; }
}

public class DeluxePizza : Pizza
{
    private decimal m_Price = 10.5M;
    public override decimal GetPrice() { return m_Price; }
}

public class SeafoodPizza : Pizza
{
    private decimal m_Price = 11.5M;
    public override decimal GetPrice() { return m_Price; }
}

// Somewhere in the code
...
  Console.WriteLine( Pizza.PizzaFactory(Pizza.Types.Seafood).GetPrice().ToString("C2") );
// $11.50
...
```

JavaScript

Pizza example:

```
//Our pizzas
function HamAndMushroomPizza(){
  var price = 8.50;
  this.getPrice = function(){
    return price;
  }
}

function DeluxePizza(){
  var price = 10.50;
  this.getPrice = function(){
    return price;
  }
}

function SeafoodPizza(){
  var price = 11.50;
  this.getPrice = function(){
    return price;
  }
}

//Pizza Factory
function PizzaFactory(){
  this.createPizza = function(type){
     switch(type){
      case "Ham and Mushroom":
        return new HamAndMushroomPizza();
      case "DeluxePizza":
        return new DeluxePizza();
      case "Seafood Pizza":
        return new SeafoodPizza();
      default:
          return new DeluxePizza();
     }
  }
}

//Usage
```

```
var pizzaPrice = new PizzaFactory().createPizza("Ham and
Mushroom").getPrice();
alert(pizzaPrice);
```

Python

```
    """
    A generic factory implementation.
    Examples:
    >>f=Factory()
    >>class A:pass
    >>f.register("createA",A)
    >>f.createA()
    <__main__.A instance at 01491E7C>

    >>> class B:
    … def __init__(self, a,b=1):
    … self.a=a
    … self.b=b
    …
    >>> f.register("createB",B,1,b=2)
    >>> f.createB()
    >>> b=f.createB()
    >>>
    >>> b.a
    1
    >>> b.b
    2

    >>> class C:
    … def __init__(self,a,b,c=1,d=2):
    … self.values = (a,b,c,d)
    …
    >>> f.register("createC",C,1,c=3)
    >>> c=f.createC(2,d=4)
    >>> c.values
    (1, 2, 3, 4)

    >>> f.register("importSerialization",__import__,"cPickle")
    >>> pickle=f.importSerialization()
    >>> pickle
    <module 'cPickle' (built-in)>
    >>> f.register("importSerialization",__import__,"marshal")
    >>> pickle=f.importSerialization()
    >>> pickle
    <module 'marshal' (built-in)>

    >>> f.unregister("importSerialization")
    >>> f.importSerialization()
    Traceback (most recent call last):
    File "<interactive input>", line 1, in ?
    AttributeError: Factory instance has no attribute 'importSerialization'
    """

    class Factory:
        def register(self, methodName, constructor, *args, **kargs):
           """register a constructor"""
           _args = [constructor]
           _args.extend(args)
           setattr(self, methodName,apply(Functor,_args, kargs))

        def unregister(self, methodName):
           """unregister a constructor"""
           delattr(self, methodName)

    class Functor:
        def __init__(self, function, *args, **kargs):
           assert callable(function), "function should be a callable obj"
           self._function = function
```

```
        self._args = args
        self._kargs = kargs

    def __call__(self, *args, **kargs):
        """call function"""
        _args = list(self._args)
        _args.extend(args)
        _kargs = self._kargs.copy()
        _kargs.update(kargs)
        return apply(self._function,_args,_kargs)
```

Prototype pattern

A **prototype pattern** is a creational design pattern used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:

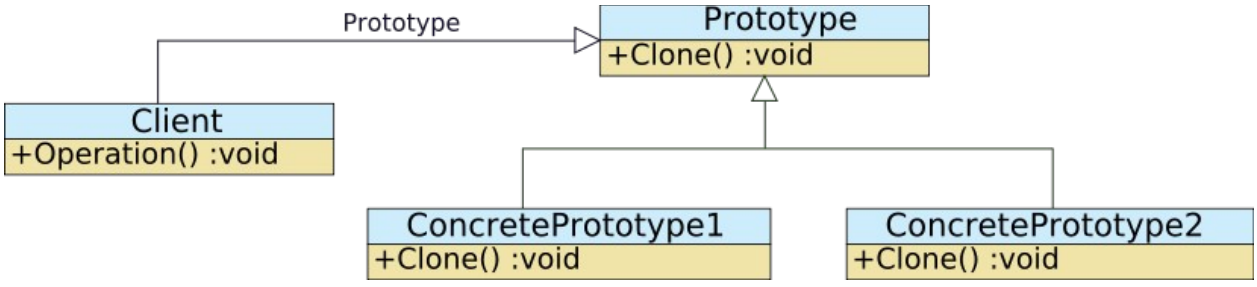
> avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
> avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

To implement the pattern, declare an abstract base class that specifies a pure virtual *clone()* method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the *clone()* operation.

The client, instead of writing code that invokes the "new" operator on a hard-wired class name, calls the *clone()* method on the prototype, calls a factory method with a parameter designating the particular concrete derived class desired, or invokes the *clone()* method through some mechanism provided by another design pattern.

Structure

The class diagram is as shown below: (the arrow from Client to Prototype should not be a triangle, but a classical arrow ->) (Prototype should be abstract)



C++ Sample code

```
#include <iostream>
#include <map>
#include <string>
```

```
#include <cstdint>

using namespace std;

enum RECORD_TYPE_en
{
  CAR,
  BIKE,
  PERSON
};

/**
 * Record is the Prototype
 */

class Record
{
  public :

    Record() {}

    virtual ~Record() {}

    virtual Record* Clone() const=0;

    virtual void Print() const=0;
};

/**
 * CarRecord is Concrete Prototype
 */

class CarRecord : public Record
{
  private :
    string m_oStrCarName;

    uint32_t m_ui32ID;

  public :

    CarRecord(const string& _oStrCarName, uint32_t _ui32ID)
      : Record(), m_oStrCarName(_oStrCarName),
        m_ui32ID(_ui32ID)
    {
    }

    CarRecord(const CarRecord& _oCarRecord)
      : Record()
    {
      m_oStrCarName = _oCarRecord.m_oStrCarName;
      m_ui32ID = _oCarRecord.m_ui32ID;
    }

    ~CarRecord() {}

    CarRecord* Clone() const
    {
      return new CarRecord(*this);
    }

    void Print() const
    {
      cout << "Car Record" << endl
        << "Name  : " << m_oStrCarName << endl
        << "Number: " << m_ui32ID << endl << endl;
    }
};


/**
```

```
 * BikeRecord is the Concrete Prototype
 */

class BikeRecord : public Record
{
  private :
    string m_oStrBikeName;

    uint32_t m_ui32ID;

  public :
    BikeRecord(const string& _oStrBikeName, uint32_t _ui32ID)
      : Record(), m_oStrBikeName(_oStrBikeName),
        m_ui32ID(_ui32ID)
    {
    }

    BikeRecord(const BikeRecord& _oBikeRecord)
      : Record()
    {
      m_oStrBikeName = _oBikeRecord.m_oStrBikeName;
      m_ui32ID = _oBikeRecord.m_ui32ID;
    }

    ~BikeRecord() {}

    BikeRecord* Clone() const
    {
      return new BikeRecord(*this);
    }

    void Print() const
    {
      cout << "Bike Record" << endl
        << "Name  : " << m_oStrBikeName << endl
        << "Number: " << m_ui32ID << endl << endl;
    }
};


/**
 * PersonRecord is the Concrete Prototype
 */

class PersonRecord : public Record
{
  private :
    string m_oStrPersonName;

    uint32_t m_ui32Age;

  public :
    PersonRecord(const string& _oStrPersonName, uint32_t _ui32Age)
      : Record(), m_oStrPersonName(_oStrPersonName),
        m_ui32Age(_ui32Age)
    {
    }

    PersonRecord(const PersonRecord& _oPersonRecord)
      : Record()
    {
      m_oStrPersonName = _oPersonRecord.m_oStrPersonName;
      m_ui32Age = _oPersonRecord.m_ui32Age;
    }

    ~PersonRecord() {}

    Record* Clone() const
    {
      return new PersonRecord(*this);
    }
```

```
    void Print() const
    {
      cout << "Person Record" << endl
        << "Name : " << m_oStrPersonName << endl
        << "Age  : " << m_ui32Age << endl << endl ;
    }
};


/**
 * RecordFactory is the client
 */

class RecordFactory
{
  private :
    map<RECORD_TYPE_en, Record* > m_oMapRecordReference;

  public :
    RecordFactory()
    {
      m_oMapRecordReference[CAR]    = new CarRecord("Ferrari", 5050);
      m_oMapRecordReference[BIKE]   = new BikeRecord("Yamaha", 2525);
      m_oMapRecordReference[PERSON] = new PersonRecord("Tom", 25);
    }

    ~RecordFactory()
    {
      delete m_oMapRecordReference[CAR];
      delete m_oMapRecordReference[BIKE];
      delete m_oMapRecordReference[PERSON];
    }

    Record* CreateRecord(RECORD_TYPE_en enType)
    {
      return m_oMapRecordReference[enType]->Clone();
    }
};

int main()
{
  RecordFactory* poRecordFactory = new RecordFactory();

  Record* poRecord;
  poRecord = poRecordFactory->CreateRecord(CAR);
  poRecord->Print();
  delete poRecord;

  poRecord = poRecordFactory->CreateRecord(BIKE);
  poRecord->Print();
  delete poRecord;

  poRecord = poRecordFactory->CreateRecord(PERSON);
  poRecord->Print();
  delete poRecord;

  delete poRecordFactory;
  return 0;
}
```

C# sample code

```
public enum RecordType
{
   Car,
   Person
}
```

```
/// <summary>
/// Record is the Prototype
/// </summary>
public abstract class Record
{
   public abstract Record Clone();
}

/// <summary>
/// PersonRecord is the Concrete Prototype
/// </summary>
public class PersonRecord : Record
{
   string name;
   int age;

   public override Record Clone()
   {
      return (Record)this.MemberwiseClone(); // default shallow copy
   }
}

/// <summary>
/// CarRecord is another Concrete Prototype
/// </summary>
public class CarRecord : Record
{
   string carname;
   Guid id;

   public override Record Clone()
   {
      CarRecord clone = (Record)this.MemberwiseClone(); // default shallow copy
      clone.id = Guid.NewGuid(); // always generate new id
      return clone;
   }
}

/// <summary>
/// RecordFactory is the client
/// </summary>
public class RecordFactory
{
   private static Dictionary<RecordType, Record> _prototypes =
      new Dictionary<RecordType, Record>();

   /// <summary>
   /// Constructor
   /// </summary>
   public RecordFactory()
   {
      _prototypes.Add(RecordType.Car, new CarRecord());
      _prototypes.Add(RecordType.Person, new PersonRecord());
   }

   /// <summary>
   /// The Factory method
   /// </summary>
   public Record CreateRecord(RecordType type)
   {
      return _prototypes[type].Clone();
   }
}
```

Java sample code

```
/** Prototype Class **/
public class Cookie implements Cloneable {
```

```
   public Object clone() {
      try {
         Cookie copy = (Cookie)super.clone();

         //In an actual implementation of this pattern you might now change references to
         //the expensive to produce parts from the copies that are held inside the
prototype.

         return copy;

      }
      catch(CloneNotSupportedException e) {
         e.printStackTrace();
         return null;
      }
   }
}

/** Concrete Prototypes to clone **/
public class CoconutCookie extends Cookie { }

/** Client Class**/
public class CookieMachine {

   private Cookie cookie;//could have been a private Cloneable cookie;

   public CookieMachine(Cookie cookie) {
      this.cookie = cookie;
   }
   public Cookie makeCookie() {
      return (Cookie)cookie.clone();
   }
   public Object clone() { }

   public static void main(String args[]) {
      Cookie tempCookie =  null;
      Cookie prot = new CoconutCookie();
      CookieMachine cm = new CookieMachine(prot);
      for (int i=0; i<100; i++)
         tempCookie = cm.makeCookie();
   }
}
```

Python

```
     from copy import deepcopy

    class Prototype:
        def __init__(self):
           self._objs = {}

        def registerObject(self, name, obj):
           """
           register an object.
           """
           self._objs[name] = obj

        def unregisterObject(self, name):
           """unregister an object"""
           del self._objs[name]

        def clone(self, name, **attr):
           """clone a registered object and add/replace attr"""
           obj = deepcopy(self._objs[name])
           obj.__dict__.update(attr)
           return obj

#################### “create another instance, w/state, of an existing instance of
unknown class/state”
```

```
from copy import deepcopy, copy

g = Graphic() # non-cooperative form
shallow_copy_of_g = copy(g)
deep_copy_of_g = deepcopy(g)

from copy import deepcopy, copy

class Graphic:
    def clone(self):
       return copy(self)
g = Graphic() # cooperative form
copy_of_g = g.clone()
```

Examples

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. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb

Sometimes creational patterns overlap - there are cases when either Prototype or Abstract Factory would be appropriate. At other times they complement each other: Abstract Factory might store a set of Prototypes from which to clone and return product objects (GoF, p126). Abstract Factory, Builder, and Prototype can use Singleton in their implementations. (GoF, p81, 134). Abstract Factory classes are often implemented with Factory Methods (creation through inheritance), but they can be implemented using Prototype (creation through delegation). (GoF, p95)

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. (GoF, p136)

Prototype doesn't require subclassing, but it does require an "initialize" operation. Factory Method requires subclassing, but doesn't require initialization. (GoF, p116)

Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well. (GoF, p126)

The rule of thumb could be that you would need to clone() an *Object* when you want to create another Object *at runtime* which is a *true copy* of the Object you are cloning. *True copy* means all the attributes of the newly created Object should be the same as the Object you are cloning. If you could have *instantiated* the class by using *new* instead, you

would get an Object with all attributes as their initial values. For example, if you are designing a system for performing bank account transactions, then you would want to make a copy of the Object which holds your account information, perform transactions on it, and then replace the original Object with the modified one. In such cases, you would want to use clone() instead of new.

Cloning in PHP

In PHP 5, unlike previous versions, objects are by default passed by reference. In order to pass by value, use the "magic function" __clone(). This makes using the prototype pattern very easy to implement. See object cloning for more information.

Singleton pattern

In software engineering, the **singleton pattern** is a design pattern that is used to restrict instantiation of a class to one object. This concept is also sometimes generalized to restrict the instance to more than one object, as for example, we can restrict the number of instances to five objects. This is useful when exactly one object is needed to coordinate actions across the system. Sometimes it is generalized to systems that operate more efficiently when only one or a few objects exist. It is also considered an anti-pattern by some people, who feel that it is overly used, introducing unnecessary limitations in situations where a sole instance of a class is not actually required. [1][2][3][4]

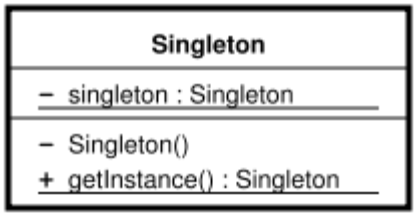
Common uses

The Abstract Factory, Builder, and Prototype patterns can use Singletons in their implementation.
Facade objects are often Singletons because only one Facade object is required.
State objects are often Singletons.
Singletons are often preferred to global variables because:

- They don't pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables.[5]
- They permit lazy allocation and initialization, where global variables in many languages will *always* consume resources.

Singletons behave differently depending on the lifetime of the virtual machine. While a software development kit may start a new virtual machine for every run which results in a new instance of the singleton being created, calls to a singleton e.g. within the virtual machine of an application server behave differently. There the virtual machine remains alive, therefore the instance of the singleton remains as well. Running the code again therefore can retrieve the "old" instance of the singleton which then may be contaminated with values in local fields which are the result of the first run.

Class diagram



Implementation

Implementation of a singleton pattern must satisfy the single instance and global access principles. It requires a mechanism to access the singleton class member without creating a class object and a mechanism to persist the value of class members among class objects. The singleton pattern is implemented by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made protected (not private, because reuse and unit test could need to access the constructor). Note the **distinction** between a simple static instance of a class and a singleton: although a singleton can be implemented as a static instance, it can also be lazily constructed, requiring no memory or resources until needed. Another notable difference is that static member classes cannot implement an interface, unless that interface is simply a marker. So if the class has to realize a contract expressed by an interface, it really has to be a singleton.

The singleton pattern must be carefully constructed in multi-threaded applications. If two threads are to execute the creation method at the same time when a singleton does not yet exist, they both must check for an instance of the singleton and then only one should create the new one. If the programming language has concurrent processing capabilities the method should be constructed to execute as a mutually exclusive operation.

The classic solution to this problem is to use mutual exclusion on the class that indicates that the object is being **instantiated**.

Example implementations

Scala

The Scala programming language supports Singleton objects out-of-the-box. The 'object' keyword creates a class and also defines a singleton object of that type.

```
object Example extends ArrayList {
    // creates a singleton called Example
}
```

Java

The Java programming language solutions provided here are all thread-safe but differ in supported language versions and lazy-loading.

The solution of Bill Pugh

This is the recommended method. It is known as the initialization on demand holder idiom and is as lazy as possible. Moreover, it works in all known versions of Java. This solution is the most portable across different Java compilers and virtual machines.

The inner class is referenced no earlier (and therefore loaded no earlier by the class loader) than the moment that getInstance() is called. Thus, this solution is thread-safe without requiring special language constructs (i.e. volatile and/or synchronized).

```
public class Singleton {
  // Protected constructor is sufficient to suppress unauthorized calls to the
constructor
  protected Singleton() {}

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

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

Traditional simple way

Just like the one above, this solution is thread-safe without requiring special language constructs, but it lacks the laziness. The INSTANCE is created as soon as the Singleton class loads. That might even be long before getInstance() is called. It might be (for example) when some static method of the class is used. If laziness is not needed or the instance needs to be created early in the application's execution, this (slightly) simpler solution can be used:

```
public class Singleton {
  public final static Singleton INSTANCE = new Singleton();

  // Protected constructor is sufficient to suppress unauthorized calls to the
constructor
  protected Singleton() {}
}
```

Sometimes the static final field is made private and a static factory-method is provided to get the instance. This way the underlying implementation may change easily while it has no more performance-issues on modern JVMs.

Java 5 solution

If and only if the compiler used is Java 5 (also known as Java 1.5) or newer, AND all Java virtual machines the application is going to run on fully support the Java 5 memory model, then (and only then) the volatile double checked locking can be used (for a detailed discussion of *why it should never be done before Java 5* see The "Double-Checked Locking is Broken" Declaration):

```
 public class Singleton {
   private static volatile Singleton INSTANCE;

   // Protected constructor is sufficient to suppress unauthorized calls to the
constructor
   protected Singleton() {}

   public static Singleton getInstance() {
      synchronized(Singleton.class) {
        if (INSTANCE == null)
          INSTANCE = new Singleton();
      }
      return INSTANCE;
   }
}
```

Allen Holub (in "Taming Java Threads", Berkeley, CA: Apress, 2000, pp. 176–178) notes that on multi-CPU systems (which are widespread as of 2007), the use of volatile may have an impact on performance approaching to that of synchronization, and raises the possibility of other problems. Thus this solution has little to recommend it over Pugh's solution described above.

The Enum-way

In the second edition of his book "Effective Java" Joshua Bloch claims that "a single-element enum type is the best way to implement a singleton"[6] for any Java that supports enums. The use of an enum is very easy to implement and has no drawbacks regarding serializable objects, which have to be circumvented in the other ways.

```
 public enum Singleton {
   INSTANCE;
 }
```

PHP 5

Singleton pattern in PHP 5[7][8]:

```
<?php
class Singleton {
  // object instance
  private static $instance;
  // The private construct prevents instantiating the class externally.  The construct
can be
  // empty, or it can contain additional instructions...
  private function __construct() {
    ...
  }
```

```
  // The clone method prevents external instantiation of copies of the Singleton class,
  // thus eliminating the possibility of duplicate classes.  The clone can be empty, or
  // it can contain additional code, most probably generating error messages in response
  // to attempts to call.
  private function __clone() {
    ...
  }
  //This method must be static, and must return an instance of the object if the object
  //does not already exist.
  public static function getInstance() {
    if (!self::$instance instanceof self) {
      self::$instance = new self;
    }
    return self::$instance;
  }
  //One or more public methods that grant access to the Singleton object, and its private
  //methods and properties via accessor methods.
  public function doAction() {
    ...
  }
}

//usage
Singleton::getInstance()->doAction();

?>
```

Actionscript 3.0

Private constructors are not available in ActionScript 3.0 - which prevents the use of the ActionScript 2.0 approach to the Singleton Pattern. Many different AS3 Singleton implementations have been published around the web.

The common method is to use a hidden key of some sort. This is used to verify that the class is only instantiated by it's getInstance() method. An error exception is then thrown if the constructor is called by a external object.

For example, in the implementation shown a private static function called "hidden" is used as the strict comparison key. This is only available to the Singleton class itself therefore ensuring that any attempts at exterior instantiation are canceled and an exception is thrown.

```
package {
        public class Singleton  {

                private static var _instance : Singleton = null;

                public function Singleton (h:Function) {
                        if !(h === hidden) {
                                throw new Error( "Singleton and can only be accessed
through Singleton.getInstance()" );
                        }
                }

                public static function getInstance() : Singleton {
                        if(_instance == null) _instance = new Singleton(hidden);
                        return _instance;
                }

                private static function hidden() : void {}

        }
}
```

Objective-C

A common way to implement a singleton in Objective-C is the following:

```
@interface MySingleton : NSObject
{
}

+ (MySingleton *)sharedSingleton;
@end

@implementation MySingleton

+ (MySingleton *)sharedSingleton
{
  static MySingleton *sharedSingleton;

  @synchronized(self)
  {
    if (!sharedSingleton)
      sharedSingleton = [[MySingleton alloc] init];

    return sharedSingleton;
  }
}

@end
```

If thread-safety is not required, the synchronization can be left out, leaving the `+sharedSingleton` method like this:

```
+ (MySingleton *)sharedSingleton
{
  static MySingleton *sharedSingleton;

  if (!sharedSingleton)
    sharedSingleton = [[MySingleton alloc] init];

  return sharedSingleton;
}
```

This pattern is widely used in the Cocoa frameworks (see for instance, `NSApplication`, `NSColorPanel`, `NSFontPanel` or `NSWorkspace`, to name but a few).

Some may argue that this is not, strictly speaking, a Singleton, because it is possible to allocate more than one instance of the object. A common way around this is to use assertions or exceptions to prevent this double allocation.

```
@interface MySingleton : NSObject
{
}

+ (MySingleton *)sharedSingleton;
@end

@implementation MySingleton

static MySingleton *sharedSingleton;

+ (MySingleton *)sharedSingleton
{
  @synchronized(self)
  {
```

```
    if (!sharedSingleton)
      [[MySingleton alloc] init];

    return sharedSingleton;
  }
}

+(id)alloc
{
  @synchronized(self)
  {
    NSAssert(sharedSingleton == nil, @"Attempted to allocate a second instance of a
singleton.");
    sharedSingleton = [super alloc];
    return sharedSingleton;
  }
}

@end
```

There are alternative ways to express the Singleton pattern in Objective-C, but they are not always as simple or as easily understood, not least because they may rely on the `-init` method returning an object other than `self`. Some of the Cocoa "Class Clusters" (e.g. `NSString`, `NSNumber`) are known to exhibit this type of behaviour.

Note that `@synchronized` is not available in some Objective-C configurations, as it relies on the NeXT/Apple runtime. It is also comparatively slow, because it has to look up the lock based on the object in parentheses. Check the history of this page for a different implementation using an `NSConditionLock`.

C++

Here is a possible implementation in C++, using the Curiously Recurring Template Pattern. In this implementation, also known as the Meyers singleton [9], the singleton is a static local object. Because C++ provides no standard multithreading support, this solution is not thread-safe in general, though some compilers (e.g. gcc) generate thread-safe code in this case.

```
#include <iostream>

template<typename T> class Singleton
{
  public:
    static T& Instance()
    {
        static T theSingleInstance;  // assumes T has a protected default constructor
        return theSingleInstance;
    }
    virtual ~Singleton();            //a virtual destructor is needed if we need to
execute
                                     //code in the derived class' destructor.
};

class OnlyOne : public Singleton<OnlyOne>
{
    friend class Singleton<OnlyOne>;
    int example_data;
  public:
    int Getexample_data() const {return example_data;}
  protected:
    OnlyOne(): example_data(42) {}   // default constructor
```

```
};


/* This test case should print "42". */
#include <iostream>
int main()
{
    std::cout << OnlyOne::Instance().Getexample_data()<<std::endl;
    return 0;
}
```

C++ (using pthreads)

A common design pattern for thread safety with the singleton class is to use double-checked locking. However, due to the ability of modern processors to re-order instructions (as long as the result is consistent with their architecturally-specified memory model), and the absence of any consideration being given to multiple threads of execution in the language standard, double-checked locking is *intrinsically prone to failure* in C++. There is no model — other than runtime libraries (e.g. POSIX threads, designed to provide concurrency primitives) — that can provide the necessary execution order.[1]

By adding a mutex to the singleton class, a thread-safe implementation may be obtained. The following is an example of double-checked locking, and as such is prone to failure.

```
#include <pthread.h>
#include <memory>
#include <iostream>

class Mutex
{
  public:
    Mutex()
    { pthread_mutex_init(&m, 0); }

    void lock()
    { pthread_mutex_lock(&m); }

    void unlock()
    { pthread_mutex_unlock(&m); }

  private:
    pthread_mutex_t m;
};

class MutexLocker
{
  public:
    MutexLocker(Mutex& pm): m(pm) { m.lock(); }
    ~MutexLocker() { m.unlock(); }
  private:
    Mutex& m;
};

class Singleton
{
  public:
    static Singleton& Instance();
    int example_data;
    ~Singleton() { }

  protected:
    Singleton(): example_data(42) { }

  private:
```

```
    static std::auto_ptr<Singleton> theSingleInstance;
    static Mutex m;
};

Singleton& Singleton::Instance()
{
    if (theSingleInstance.get() == 0)
    {
      MutexLocker obtain_lock(m);
      if (theSingleInstance.get() == 0)
      {
        theSingleInstance.reset(new Singleton);
      }
    }
    return *theSingleInstance;
}

std::auto_ptr<Singleton> Singleton::theSingleInstance;
Mutex Singleton::m;

int main()
{
    std::cout << Singleton::Instance().example_data << std::endl;
    return 0;
}
```

Note the use of the `MutexLocker` class in the `Singleton::Instance()` function. The `MutexLocker` is being used as an RAII object, also known as *scoped lock*, guaranteeing that the mutex lock will be relinquished even if an exception is thrown during the execution of `Singleton::Instance()`, since the language specification pledges that the destructors of automatically allocated objects are invoked during stack unwind.

This implementation invokes the mutex-locking primitives for each call to `Singleton::Instance()`, even though the mutex is only needed once, the first time the method is called. To get rid of the extra mutex operations, the programmer can explicitly construct `Singleton::theSingleInstance` early in the program (say, in `main`); or, the class can be optimized (in this case, using `pthread_once`) to cache the value of the initialized pointer in thread-local storage.

More code must be written if the singleton code is located in a static library, but the program is divided into DLLs.[*citation needed*] Each DLL that uses the singleton will create a new and distinct instance of the singleton.[*citation needed*] To avoid that, the singleton code must be linked in a DLL. Alternatively, the singleton can be rewritten to use a memory-mapped file to store `theSingleInstance`.

C#

The simplest of all is:

```
public class Singleton
{
    // The combination of static and readonly makes the instantiation
    // thread safe.  Plus the constructor being protected (it can be
    // private as well), makes the class sure to not have any other
    // way to instantiate this class than using this member variable.
    public static readonly Singleton Instance = new Singleton();

    // Protected constructor is sufficient to avoid other instantiation
```

```
    // This must be present otherwise the compiler provides a default
    // public constructor
    //
    protected Singleton()
    {
    }
}
```

This example is thread-safe with lazy initialization.

```
/// Class implements singleton pattern.

public class Singleton
{
        // Protected constructor is sufficient to avoid other instantiation
        // This must be present otherwise the compiler provides
        // a default public constructor
        protected Singleton()
        {
        }


        /// Return an instance of <see cref="Singleton"/>

        public static Singleton Instance
        {
            get
            {
                /// An instance of Singleton wont be created until the very first
                /// call to the sealed class. This is a CLR optimization that
                /// provides a properly lazy-loading singleton.
                return SingletonCreator.CreatorInstance;
            }
        }


        /// Sealed class to avoid any heritage from this helper class

        private sealed class SingletonCreator
        {
          // Retrieve a single instance of a Singleton
          private static readonly Singleton _instance = new Singleton();


          /// Return an instance of the class <see cref="Singleton"/>

          public static Singleton CreatorInstance
          {
            get { return _instance; }
          }
        }
}
```

Another example(using static constructor)

```
using System;
using System.Threading;

namespace Singleton1
{
        class Singleton
        {
                private static readonly Singleton _instance;
                private int v;


                /// Protected constructor is sufficient to prevent
                /// instantiation by using 'new' keyword.
```

```
		protected Singleton()
		{
			Console.WriteLine("Singleton Instance Creating...");
			this.V = 0;

			Thread.Sleep(1000); // Simulate a HEAVY creation cost.

			Console.WriteLine("Singleton Instance Created.");
		}


		/// Static constructor

		static Singleton()
		{
			_instance = new Singleton();
		}

		public static Singleton Instance
		{
			get
			{
				return _instance;
			}
		}

		public int V
		{
			get { return v; }
			set { v = value; }
		}

		public void DoSomeWork()
		{
			Console.Write("#");
			lock(this)
			{
				V++;
			}
			Thread.Sleep(500);
		}
	}

	class TestClass
	{

		/// Singleton with Multithread

		static void Multithread()
		{
			Singleton instance = Singleton.Instance;

			Thread t = new Thread(new ThreadStart(instance.DoSomeWork));
			t.Start();
		}

		static void Main(string[] args)
		{
			int i;

			for(i = 0; i < 10; i++)
			{
				Console.WriteLine("Do some work...");
				Thread.Sleep(100);
			}

			// ^- Until now, this application has no any singleton instance.
-^ //

			for(i = 0; i < 300; i++)
```

```
                        {
                                // At first time, Singleton class make an instance of
itself.
                                // Other time, it will return absolutly same instance.
                                Multithread();
                        }

                        Thread.Sleep(1000); // Wait for sure all threads finished.

                        Console.WriteLine("");
                        Console.WriteLine("V value - expected: {0}, actual:
{1}",i,Singleton.Instance.V);

                        // Singleton with 'new' is Strictly forbidden.

                        // Singleton y = new Singleton(); // No acceptable.
                }
        }
}
```

Example in C# 2.0 (thread-safe with lazy initialization) Note: This is not a recommended implementation because "TestClass" has a default public constructor, and that violates the definition of a Singleton. A proper Singleton must never be instantiable more than once.

```
/// Parent for singleton

/// <typeparam name="T">Singleton class</typeparam>
  public class Singleton<T> where T : class, new()
  {

    protected Singleton() { }

    private sealed class SingletonCreator<S> where S : class, new()
    {
      private static readonly S instance = new S();

      public static S CreatorInstance
      {
        get { return instance; }
      }
    }

    public static T Instance
    {
      get { return SingletonCreator<T>.CreatorInstance; }
    }

  }


/// Concrete Singleton

public class TestClass : Singleton<TestClass>
{
    public string TestProc()
    {
        return "Hello World";
    }
}

// Somewhere in the code
.....
TestClass.Instance.TestProc();
.....
```

Python

According to influential Python programmer Alex Martelli, *The Singleton design pattern (DP) has a catchy name, but the wrong focus—on identity rather than on state. The Borg design pattern has all instances share state instead.*[10] A rough consensus in the Python community is that sharing state among instances is more elegant, at least in Python, than is caching creation of identical instances on class initialization. Coding shared state is nearly transparent:

```
class Borg:
   __shared_state = {}
   def __init__(self):
       self.__dict__ = self.__shared_state
   # and whatever else is needed in the class -- that's all!
```

But with the new style class, this is a better solution, because only one instance is created:

```
class  Singleton (object):
    instance = None
    def __new__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = object.__new__(cls, *args, **kwargs)
        return cls.instance

#Usage
mySingleton1 =  Singleton()
mySingleton2 =  Singleton()

#mySingleton1 and  mySingleton2 are the same instance.
assert mySingleton1 is mySingleton2
```

Two caveats:

> The *__init__*-method is called every time *Singleton()* is called, unless *cls.__init__* is set to an empty function.
> If it is needed to inherit from the *Singleton*-class, *instance* should probably be a *dictionary* belonging explicitly to the *Singleton*-class.

```
class  InheritableSingleton (object):
    instances = {}
    def __new__(cls, *args, **kwargs):
        if InheritableSingleton.instances.get(cls) is None:
            cls.__original_init__ = cls.__init__
            InheritableSingleton.instances[cls] = object.__new__(cls,
*args, **kwargs)
        elif cls.__init__ == cls.__original_init__:
            def nothing(*args, **kwargs):
                pass
            cls.__init__ = nothing
        return InheritableSingleton.instances[cls]
```

To create a singleton that inherits from a non-singleton, multiple inheritance must be used.

```
class  Singleton (NonSingletonClass, object):
    instance = None
    def __new__(cls, *args, **kargs):
        if cls.instance is None:
            cls.instance = object.__new__(cls, *args, **kargs)
        return cls.instance
```

Be sure to call the NonSingletonClass's __init__ function from the Singleton's __init__ function.

Perl

In a Perl version equal or superior to 5.10 a state variable can be used.

```
package MySingletonClass;
use strict;
use warnings;
use 5.10;

sub new {
    my ($class) = @_;
    state $the_instance;

    if (! defined $the_instance) {
        $the_instance = bless { }, $class;
    }
    return $the_instance;
}
```

In older Perls, just use a closure.

```
package MySingletonClass;
use strict;
use warnings;

my $THE_INSTANCE;
sub new {
    my ($class) = @_;

    if (! defined $THE_INSTANCE) {
        $THE_INSTANCE = bless { }, $class;
    }
    return $THE_INSTANCE;
}
```

If Moose is used, there is the MooseX::Singleton extension module.

Ruby

In Ruby, just include the Singleton in the class.

```
require 'singleton'

class Example
```

```
  include Singleton
end
```

ABAP Objects

In ABAP Objects, to make instantiation private, add an attribute of type ref to the class, and a static method to control instantiation.

```
program pattern_singleton.

*---------------------------------------------------------------------*

   * Singleton
   * =========
   * Intent

*

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

*---------------------------------------------------------------------*

class lcl_Singleton definition create private.

  public section.

  class-methods:
    get_Instance returning value(Result) type ref to lcl_Singleton.

  private section.
    class-data:
      fg_Singleton type ref to lcl_Singleton.

endclass.

class lcl_Singleton implementation.

  method get_Instance.
    if ( fg_Singleton is initial ).
      create object fg_Singleton.
    endif.
    Result = fg_Singleton.
  endmethod.

endclass.
```

Prototype-based singleton

In a prototype-based programming language, where objects but not classes are used, a "singleton" simply refers to an object without copies or that is not used as the prototype for any other object. Example in Io:

```
Foo := Object clone
Foo clone := Foo
```

Example of use with the factory method pattern

The singleton pattern is often used in conjunction with the factory method pattern to create a system-wide resource whose specific type is not known to the code that uses it. An example of using these two patterns together is the Java Abstract Windowing Toolkit (AWT).

`java.awt.Toolkit` is an abstract class that binds the various AWT components to particular native toolkit implementations. The `Toolkit` class has a `Toolkit.getDefaultToolkit()` factory method that returns the platform-specific subclass of `Toolkit`. The `Toolkit` object is a singleton because the AWT needs only a single object to perform the binding and the object is relatively expensive to create. The toolkit methods must be implemented in an object and not as static methods of a class because the specific implementation is not known by the platform-independent components. The name of the specific `Toolkit` subclass used is specified by the "`awt.toolkit`" environment property accessed through `System.getProperties()`.

The binding performed by the toolkit allows, for example, the backing implementation of a `java.awt.Window` to bound to the platform-specific `java.awt.peer.WindowPeer` implementation. Neither the `Window` class nor the application using the window needs to be aware of which platform-specific subclass of the peer is used.

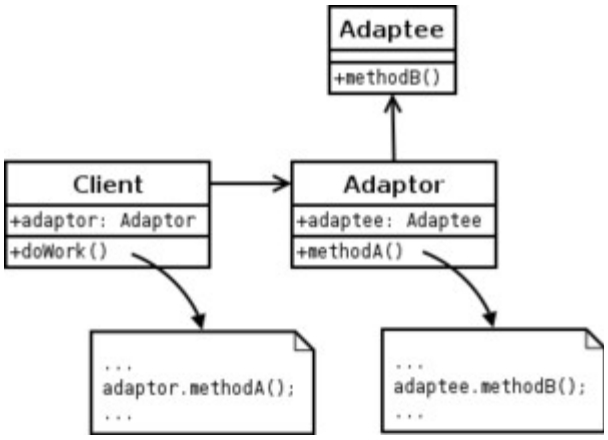
Adapter pattern

In computer programming, the **adapter design pattern** (often referred to as the **wrapper pattern** or simply a **wrapper**) translates one interface for a class into a compatible interface. An *adapter* allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients whilst utilizing the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small. The adapter is also responsible for transforming data into appropriate forms. For instance, if multiple boolean values are stored as a single integer but your consumer requires a 'true'/'false', the adapter would be responsible for extracting the appropriate values from the integer value.

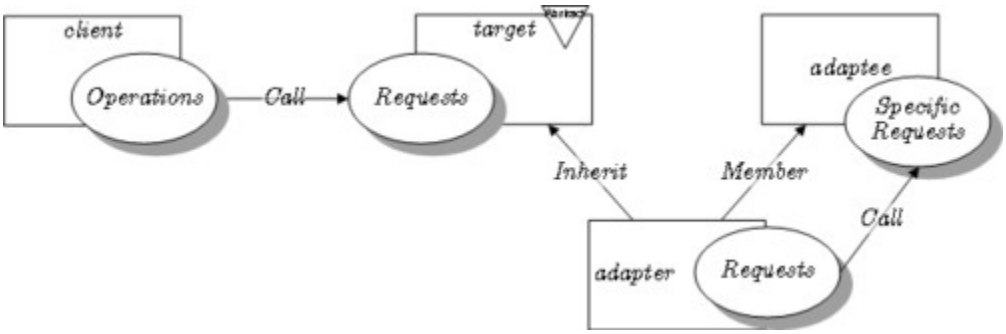
Structure

There are two types of adapter patterns:

Object Adapter pattern
: In this type of adapter pattern, the adapter contains an instance of the class it wraps. In this situation, the adapter makes calls to the instance of the wrapped object.



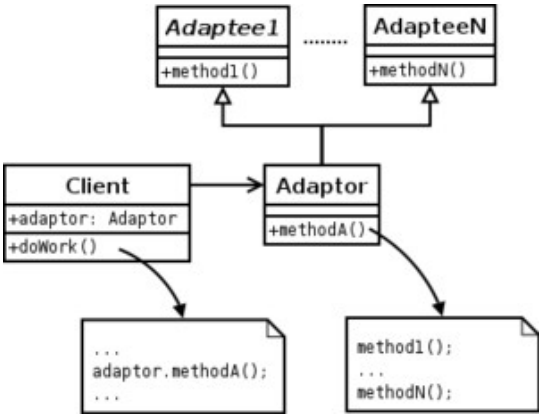
The object adapter pattern expressed in UML. The adapter *hides* the adaptee's interface from the client.



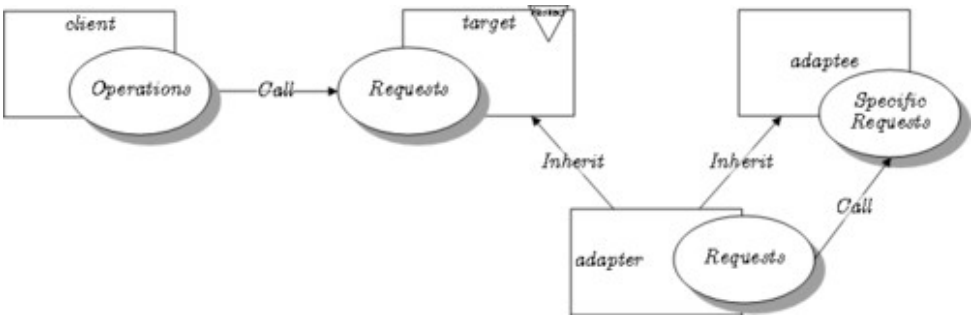
The object adapter pattern expressed in LePUS3.

Class Adapter pattern

This type of adapter uses multiple inheritance to achieve its goal. The adapter is created inheriting from both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class, especially in languages such as Java that do not support multiple inheritance.



The class adapter pattern expressed in UML.



The class adapter pattern expressed in LePUS3

The adapter pattern is useful in situations where an already existing class provides some or all of the services you need but does not use the interface you need. A good real life example is an adapter that converts the interface of a Document Object Model of an XML document into a tree structure that can be displayed. A link to a tutorial that uses the adapter design pattern is listed in the links below.

Sample - Object Adapter

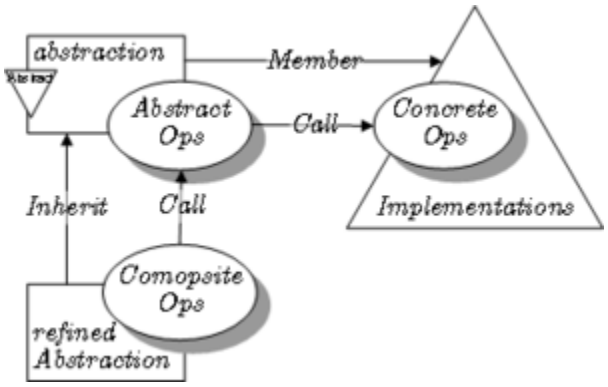
```
# Python code sample
class Target(object):
    def specific_request(self):
        return 'Hello Adapter Pattern!'

class Adapter(object):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return self.adaptee.specific_request()

client = Adapter(Target())
print client.request()
```

Sample - Class Adapter

```
/**
 * Java code sample
 */

/* the client class should instantiate adapter objects */
/* by using a reference of this type */
interface Stack<T>
{
  void push (T o);
  T pop ();
  T top ();
}

/* DoubleLinkedList is the adaptee class */
class DList<T>
{
  public void insert (DNode pos, T o) { ... }
  public void remove (DNode pos) { ... }

  public void insertHead (T o) { ... }
  public void insertTail (T o) { ... }
```

```
  public T removeHead () { ... }
  public T removeTail () { ... }

  public T getHead () { ... }
  public T getTail () { ... }
}

/* Adapt DList class to Stack interface is the adapter class */
class DListImpStack<T> extends DList<T> implements Stack<T>
{
  public void push (T o) {
    insertTail (o);
  }

  public T pop () {
    return removeTail ();
  }

  public T top () {
    return getTail ();
  }
}
```

Bridge pattern



Bridge in LePUS3 (legend)

The **bridge pattern** is a design pattern used in software engineering which is meant to *"decouple an abstraction from its implementation so that the two can vary independently"* (Gamma et al.). The *bridge* uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program. The bridge pattern is useful when not only the class itself varies often but also what the class does. The class itself can be thought of as the *implementation* and what the class can do as the *abstraction*.

Variant: The implementation can be decoupled even more by deferring the presence of the implementation to the point where the abstraction is utilized (as illustrated by the Visual Prolog example below).

Non-technical examples

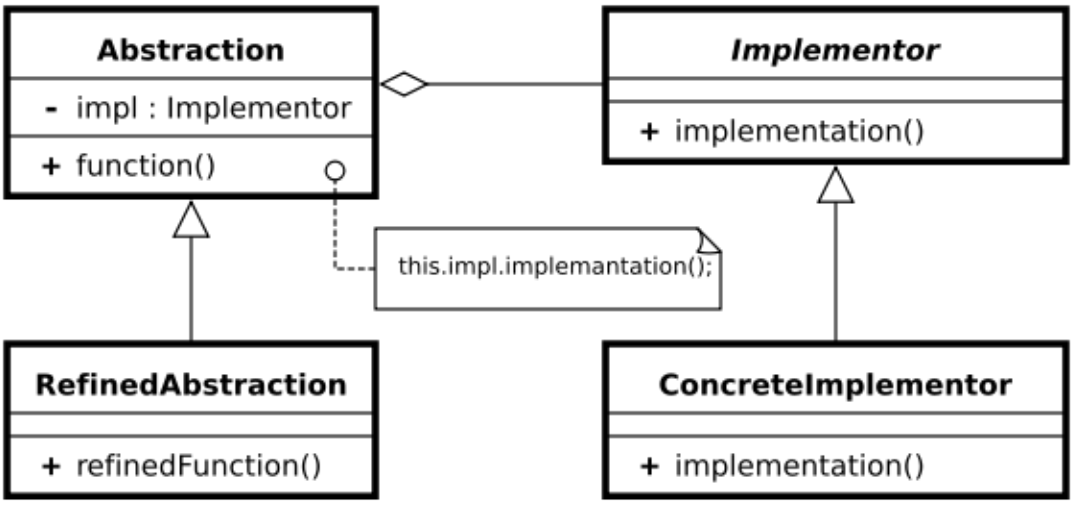
Shape abstraction

When the abstraction and implementation are separated, they can vary independently. Consider the abstraction of shapes. There are many types of shapes, each with its own properties. And there are things that all shapes do. One thing all shapes can do is draw themselves. However, drawing graphics to a screen can sometimes be dependent on different graphics implementations or operating systems. Shapes have to be able to be drawn on many types of operating systems. Having the shape itself implement them all, or modifying the shape class to work with different architectures is not practical. The bridge helps by allowing the creation of new implementation classes that provide the drawing implementation. The abstraction class, shape, provides methods for getting the size or properties of a shape. The implementation class, drawing, provides an interface for drawing graphics. If a new shape needs to be created or there is a new graphics API to be drawn on, then it is very easy to add a new implementation class that implements the needed features.[1]

Car abstraction

Imagine two types of cars (the *abstraction*), a Jaguar and a Mercedes (both are *Refinements of the Abstraction*). The *Abstraction* defines that a Car has features such as tires and an engine. *Refinements of the Abstraction* declare what specific kind of tires and engine it has.
Finally, there are two types of road. The *road* is the *Implementor* (see image below). A highway and an interstate highway are the *Implementation Details*. Any car refinement needs to be able to drive on any type of road; this concept is what the Bridge Pattern is all about.

Structure



Abstraction
: defines the abstract interface
: maintains the Implementor reference

Refined Abstraction
: extends the interface defined by Abstraction

Implementor
: defines the interface for implementation classes

ConcreteImplementor
: implements the Implementor interface

Code examples

Java

The following Java (SE 6) program illustrates the 'shape' example given above and will output:

```
API1.circle at 1.000000:2.000000 radius 7.500000
API2.circle at 5.000000:7.000000 radius 27.500000
/** "Implementor" */
interface DrawingAPI
{
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor" 1/2 */
class DrawingAPI1 implements DrawingAPI
{
   public void drawCircle(double x, double y, double radius)
   {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
   }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI
```

```
{
   public void drawCircle(double x, double y, double radius)
   {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
   }
}

/** "Abstraction" */
interface Shape
{
   public void draw();                                              // low-level
   public void resizeByPercentage(double pct);     // high-level
}

/** "Refined Abstraction" */
class CircleShape implements Shape
{
   private double x, y, radius;
   private DrawingAPI drawingAPI;
   public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI)
   {
       this.x = x;  this.y = y;  this.radius = radius;
       this.drawingAPI = drawingAPI;
   }

   // low-level i.e. Implementation specific
   public void draw()
   {
        drawingAPI.drawCircle(x, y, radius);
   }
   // high-level i.e. Abstraction specific
   public void resizeByPercentage(double pct)
   {
        radius *= pct;
   }
}

/** "Client" */
class BridgePattern {
   public static void main(String[] args)
   {
       Shape[] shapes = new Shape[2];
       shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
       shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

       for (Shape shape : shapes)
       {
           shape.resizeByPercentage(2.5);
           shape.draw();
       }
   }
}
```

C#

The following C# program illustrates the "shape" example given above and will output:

```
API1.circle at 1:2 radius 7.5
API2.circle at 5:7 radius 27.5
 using System;

 /** "Implementor" */
 interface IDrawingAPI {
    void DrawCircle(double x, double y, double radius);
 }

 /** "ConcreteImplementor" 1/2 */
 class DrawingAPI1 : IDrawingAPI {
    public void DrawCircle(double x, double y, double radius)
```

```
    {
        System.Console.WriteLine("API1.circle at {0}:{1} radius {2}", x, y, radius);
    }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 : IDrawingAPI
{
    public void DrawCircle(double x, double y, double radius)
    {
        System.Console.WriteLine("API2.circle at {0}:{1} radius {2}", x, y, radius);
    }
}

/** "Abstraction" */
interface IShape {
    void Draw();                              // low-level (i.e. Implementation-specific)
    void ResizeByPercentage(double pct);      // high-level (i.e. Abstraction-specific)
}

/** "Refined Abstraction" */
class CircleShape : IShape {
    private double x, y, radius;
    private IDrawingAPI drawingAPI;
    public CircleShape(double x, double y, double radius, IDrawingAPI drawingAPI)
    {
        this.x = x;  this.y = y;  this.radius = radius;
        this.drawingAPI = drawingAPI;
    }
    // low-level (i.e. Implementation-specific)
    public void Draw() { drawingAPI.DrawCircle(x, y, radius); }
    // high-level (i.e. Abstraction-specific)
    public void ResizeByPercentage(double pct) { radius *= pct; }
}

/** "Client" */
class BridgePattern {
    public static void Main(string[] args) {
        IShape[] shapes = new IShape[2];
        shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
        shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

        foreach (IShape shape in shapes) {
            shape.ResizeByPercentage(2.5);
            shape.Draw();
        }
    }
}
```

C# using generics

The following C# program illustrates the "shape" example given above and will output:

```
API1.circle at 1:2 radius 7.5
API2.circle at 5:7 radius 27.5
```

```
 using System;

 /** "Implementor" */
 interface IDrawingAPI {
    void DrawCircle(double x, double y, double radius);
 }

 /** "ConcreteImplementor" 1/2 */
 struct DrawingAPI1 : IDrawingAPI {
    public void DrawCircle(double x, double y, double radius)
    {
        System.Console.WriteLine("API1.circle at {0}:{1} radius {2}", x, y, radius);
    }
 }
```

```
/** "ConcreteImplementor" 2/2 */
struct DrawingAPI2 : IDrawingAPI
{
   public void DrawCircle(double x, double y, double radius)
   {
       System.Console.WriteLine("API2.circle at {0}:{1} radius {2}", x, y, radius);
   }
}

/** "Abstraction" */
interface IShape {
   void Draw();                              // low-level (i.e. Implementation-specific)
   void ResizeByPercentage(double pct);      // high-level (i.e. Abstraction-specific)
}

/** "Refined Abstraction" */
class CircleShape<T> : IShape
   where T : struct, IDrawingAPI
{
   private double x, y, radius;
   private static IDrawingAPI drawingAPI = new T();
   public CircleShape(double x, double y, double radius)
   {
       this.x = x;  this.y = y;  this.radius = radius;
   }
   // low-level (i.e. Implementation-specific)
   public void Draw() { drawingAPI.DrawCircle(x, y, radius); }
   // high-level (i.e. Abstraction-specific)
   public void ResizeByPercentage(double pct) { radius *= pct; }
}

/** "Client" */
class BridgePattern {
   public static void Main(string[] args) {
       IShape[] shapes = new IShape[2];
       shapes[0] = new CircleShape<DrawingAPI1>(1, 2, 3);
       shapes[1] = new CircleShape<DrawingAPI2>(5, 7, 11);

       foreach (IShape shape in shapes) {
           shape.ResizeByPercentage(2.5);
           shape.Draw();
       }
   }
}
```

C++

The following C++ program illustrates the "shape" example given above and will output:

```
API1.circle at 1:2 7.5
API2.circle at 5:7 27.5
#include <iostream>

using namespace std;

/* Implementor*/
class DrawingAPI {
  public:
   virtual void drawCircle(double x, double y, double radius) = 0;
   virtual ~DrawingAPI(){};
};

/* Concrete ImplementorA*/
class DrawingAPI1 : public DrawingAPI {
```

```
  public:
   void drawCircle(double x, double y, double radius) {
      cout << "API1.circle at " << x << ":" << y << " " << radius << endl;
   }
};

/* Concrete ImplementorB*/
class DrawingAPI2 : public DrawingAPI {
public:
   void drawCircle(double x, double y, double radius) {
      cout << "API2.circle at " << x << ":" << y << " " <<  radius << endl;
   }
};

/* Abstraction*/
class Shape {
  public:
   virtual ~Shape() {};
   virtual void draw() = 0;
   virtual void resizeByPercentage(double pct) = 0;
};

/* Refined Abstraction*/
class CircleShape : public Shape {
  public:
   CircleShape(double x, double y,double radius, DrawingAPI *drawingAPI){
      m_x = x;
      m_y = y;
      m_radius = radius;
      m_drawingAPI = drawingAPI;
   }
   void draw() {
      m_drawingAPI->drawCircle(m_x,m_y,m_radius);
   }
   void resizeByPercentage(double pct) {
      m_radius *= pct;
   }
  private:
   double m_x,m_y,m_radius;
   DrawingAPI *m_drawingAPI;
};

int main(void) {
   DrawingAPI1 dap1;
   DrawingAPI2 dap2;
   CircleShape circle1(1,2,3,&dap1);
   CircleShape circle2(5,7,11,&dap2);
   circle1.resizeByPercentage(2.5);
   circle2.resizeByPercentage(2.5);
   circle1.draw();
   circle2.draw();
   return 0;
}
```

Composite pattern

In computer science, the **composite pattern** is a partitioning design pattern. Composite allows a group of objects to be treated in the same way as a single instance of an object. The intent of composite is to "compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly."[1]

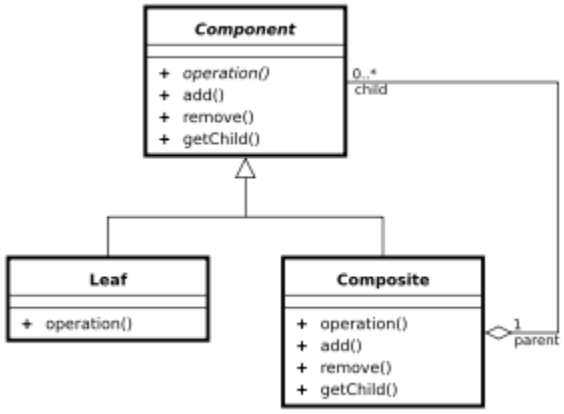
Motivation

When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly. In object-oriented programming, a composite is an object (e.g., a shape) designed as a composition of one-or-more similar objects (other kinds of shapes/geometries), all exhibiting similar functionality. This is known as a "has-a" relationship between objects. The key concept is that you can manipulate a single instance of the object just as you would a group of them. The operations you can perform on all the composite objects often have a least common denominator relationship. For example, if defining a system to portray grouped shapes on a screen, it would be useful to define resizing a group of shapes to have the same effect (in some sense) as resizing a single shape.

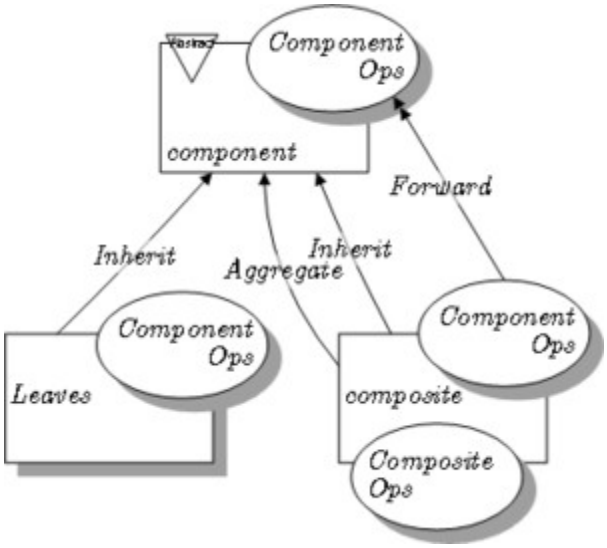
When to use

Composite can be used when clients should ignore the difference between compositions of objects and individual objects.[1] If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice; it is less complex in this situation to treat primitives and composites as homogeneous.

Structure



Composite pattern in UML.



Composite pattern in LePUS3.

Component

is the abstraction for all components, including composite ones
declares the interface for objects in the composition
implements default behavior for the interface common to all classes, as appropriate
declares an interface for accessing and managing its child components
(optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

Leaf

represents leaf objects in the composition
implements all Component methods

Composite

represents a composite Component (component having children)
implements methods to manipulate children
implements all Component methods, generally by delegating them to its children

Java

The following example, written in Java, implements a graphic class, which can be either an ellipse or a composition of several graphics. Every graphic can be printed. In algebraic form,

```
Graphic = ellipse | GraphicList
GraphicList = empty | ellipse GraphicList
```

It could be extended to implement several other shapes (rectangle, etc.) and methods (translate, etc.).

List = empty_list | atom List | List List

```
import java.util.List;
import java.util.ArrayList;

/** "Component" */
interface Graphic {

    //Prints the graphic.
    public void print();

}

/** "Composite" */
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> mChildGraphics = new ArrayList<Graphic>();

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        mChildGraphics.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        mChildGraphics.remove(graphic);
    }

}


/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }

}


/** Client */
public class Program {

    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();
```

```
        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (four times the string "Ellipse").
        graphic.print();
    }
}
```

C++ Example

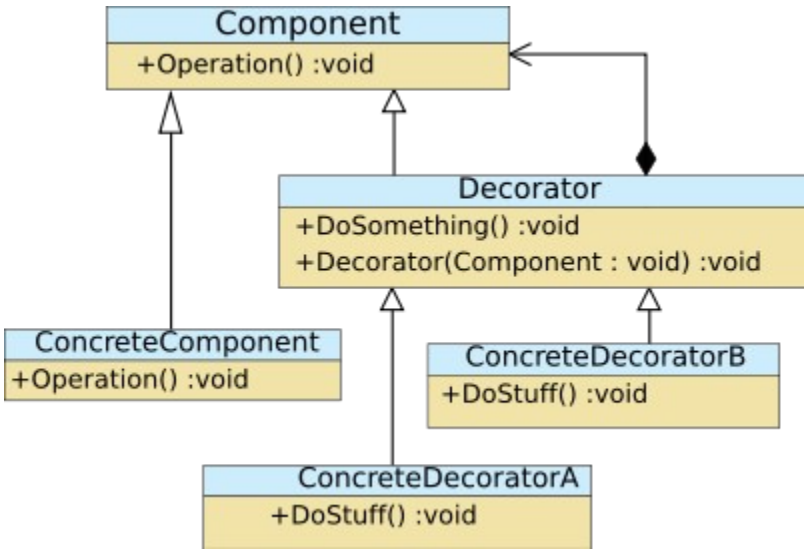
```
Container 1:
   0
   1
Container 2:
   2
   3
   4
#include <iostream>
#include <vector>
#include <string>

using std::cout;
using std::vector;
using std::string;

class Component
{
  public:
   virtual void list() const = 0;
   virtual ~Component(){};
};

class Leaf : public Component
{
  public:
   explicit Leaf(int val) : value_(val)
   {
   }
   void list() const
   {
      cout << "   " << value_ << "\n";
   }
  private:
   int value_;
};

class Composite : public Component
{
  public:
   explicit Composite(string id) : id_(id)
   {
   }
   void add(Component *obj)
   {
      table_.push_back(obj);
   }
   void list() const
   {
      cout << id_ << ":" << "\n";
      for (vector<Component*>::const_iterator it = table_.begin();
       it != table_.end(); ++it)
      {
         (*it)->list();
      }
```

```
    }
  private:
    vector <Component*> table_;
    string id_;
};

int main()
{
    Leaf num0(0);
    Leaf num1(1);
    Leaf num2(2);
    Leaf num3(3);
    Leaf num4(4);
    Composite container1("Container 1");
    Composite container2("Container 2");

    container1.add(&num0);
    container1.add(&num1);

    container2.add(&num2);
    container2.add(&num3);
    container2.add(&num4);

    container1.add(&container2);
    container1.list();
    return 0;
}
```

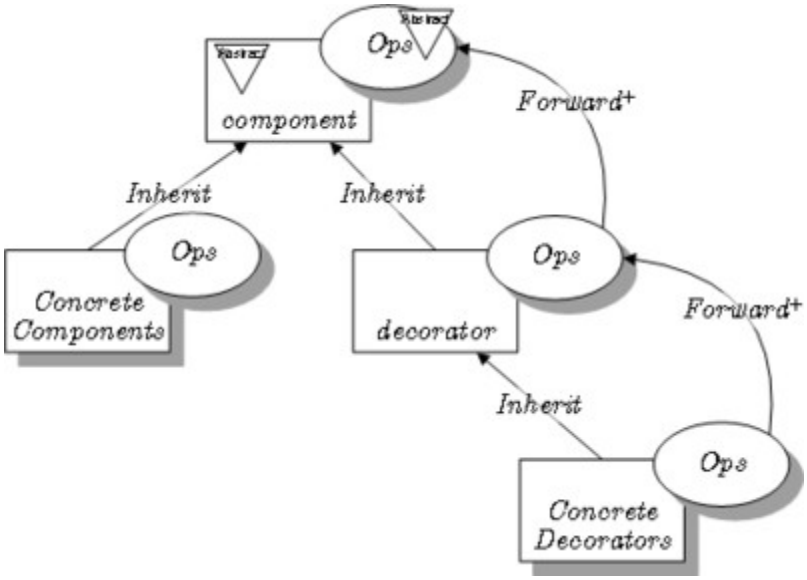
Decorator pattern

For decorators in Python, see Python syntax and semantics#Decorators.

In object-oriented programming, the **decorator pattern** is a design pattern that allows new/additional behaviour to be added to an existing class dynamically.



UML Class diagram of the decorator pattern



LePUS3 chart of the decorator pattern

Introduction

The decorator pattern can be used to make it possible to extend (decorate) the functionality of a class at runtime. This works by adding a new *decorator* class that wraps the original class. This wrapping is typically achieved by passing the original object as a parameter to the constructor of the decorator when it is created. The decorator implements the new functionality, but for functionality that is not new, the original (wrapped) class is used. The decorating class must have the same interface as the original class.

The decorator pattern is an alternative to subclassing. Subclassing adds behaviour at compile time whereas decorating can provide new behaviour at runtime.

This difference becomes most important when there are several *independent* ways of extending functionality. In some object-oriented programming languages, classes cannot be created at runtime, and it is typically not possible to predict what combinations of extensions will be needed at design time. This would mean that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis. An example of the decorator pattern is the Java I/O Streams implementation.

Motivation

As an example, consider a window in a windowing system. To allow scrolling of the window's contents, we may wish to add horizontal or vertical scrollbars to it, as appropriate. Assume windows are represented by instances of the *Window* class, and assume this class has no functionality for adding scrollbars. We could create a subclass

ScrollingWindow that provides them, or we could create a *ScrollingWindowDecorator* that merely adds this functionality to existing *Window* objects. At this point, either solution would be fine.

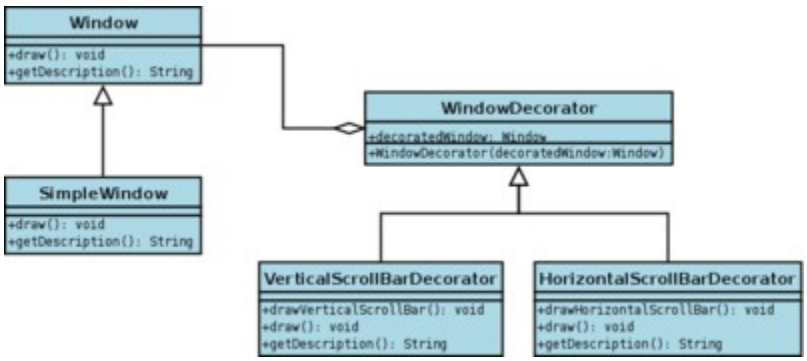
Now let's assume we also wish the option to add borders to our windows. Again, our original *Window* class has no support. The *ScrollingWindow* subclass now poses a problem, because it has effectively created a new kind of window. If we wish to add border support to *all* windows, we must create subclasses *WindowWithBorder* and *ScrollingWindowWithBorder*. Obviously, this problem gets worse with every new feature to be added. For the decorator solution, we need merely create a new *BorderedWindowDecorator*—at runtime, we can decorate existing windows with the *ScrollingWindowDecorator* or the *BorderedWindowDecorator* or both, as we see fit.

Another good example of where a decorator can be desired is when there is a need to restrict access to an object's properties or methods according to some set of rules or perhaps several parallel sets of rules (different user credentials, etc). In this case instead of implementing the access control in the original object it is left unchanged and unaware of any restrictions on its use, and it is wrapped in an access control decorator object, which can then serve only the permitted subset of the original object's interface.

Applicability

Consider viewing a webpage with a web browser. The webpage itself displays the information needed, but the web browser knows nothing about the content. It is likely that the webpage doesn't fit in the entire browser area and a scrollbar is required to show the information. The web browser doesn't need to assume that all webpages will require a scrollbar and it certainly should never assume a scrollbar is never needed. Browsers will typically display the scrollbar only if it is necessary and hide it if it is unnecessary. In this case, the scrollbar is the "decoration" to the webpage. It takes care of whether it should be displayed dynamically as opposed to statically forcing the webpage display to be a subclass of the scrollbar display. Thus, it is up to the scrollbar to decide whether it should display itself (instead of trying to force that responsibility on the webpage or on the external parts of the web browser).

Example




UML Diagram for the Window Example

This Java example uses the window/scrolling scenario.

```
// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription(); // returns a description of the Window
}


// implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```

The following classes contain the decorators for all `Window` classes, including the decorator classes themselves..

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow; // the Window being decorated

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
}


// the first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }

    public void draw() {
        drawVerticalScrollBar();
        decoratedWindow.draw();
    }

    private void drawVerticalScrollBar() {
        // draw the vertical scrollbar
    }

    public String getDescription() {
        return decoratedWindow.getDescription() + ", including vertical scrollbars";
    }
}


// the second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }

    public void draw() {
        drawHorizontalScrollBar();
        decoratedWindow.draw();
    }
```

```
    private void drawHorizontalScrollBar() {
        // draw the horizontal scrollbar
    }

    public String getDescription() {
        return decoratedWindow.getDescription() + ", including horizontal scrollbars";
    }
}
```

Here's a test program that creates a `Window` instance which is fully decorated (i.e., with vertical and horizontal scrollbars), and prints its description:

```
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator(new SimpleWindow()));

        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

The output of this program is "simple window, including vertical scrollbars, including horizontal scrollbars". Notice how the `getDescription` method of the two decorators first retrieve the decorated `Window`'s description and *decorates* it with a suffix.

In C++ we have:

```
#include <iostream>

using namespace std;

/* Component (interface) */
class Widget {

public:
  virtual void draw() = 0;
  virtual ~Widget() {}
};

/* ConcreteComponent */
class TextField : public Widget {

private:
   int width, height;

public:
   TextField( int w, int h ){
      width  = w;
      height = h;
   }

   void draw() {
      cout << "TextField: " << width << ", " << height << '\n';
   }
};

/* Decorator (interface) */
class Decorator : public Widget {

private:
   Widget* wid;       // reference to Widget
```

```
public:
   Decorator( Widget* w )  {
     wid = w;
   }

   void draw() {
     wid->draw();
   }

   ~Decorator() {
     delete wid;
   }
};

/* ConcreteDecoratorA */
class BorderDecorator : public Decorator {

public:
   BorderDecorator( Widget* w ) : Decorator( w ) { }
   void draw() {
      Decorator::draw();
      cout << "   BorderDecorator" << '\n';
   }
};

/* ConcreteDecoratorB */
class ScrollDecorator : public Decorator {
public:
   ScrollDecorator( Widget* w ) : Decorator( w ) { }
   void draw() {
      Decorator::draw();
      cout << "   ScrollDecorator" << '\n';
   }
};

int main( void ) {

   Widget* aWidget = new BorderDecorator(
                     new ScrollDecorator(
                     new TextField( 80, 24 )));
   aWidget->draw();
   delete aWidget;
}
```

In C#:

```
namespace GSL_Decorator_pattern
{
        interface IWindowObject
        {
                void draw(); // draws the object
                String getDescription(); // returns a description of the object
        }


        class ControlComponent : IWindowObject
        {
                public ControlComponent()
                {
                }

                public void draw() // draws the object
                {
                        Console.WriteLine( "ControlComponent::draw()" );
                }

                public String getDescription() // returns a description of the object
                {
                        return "ControlComponent::getDescription()";
```

```
                }
        }

        abstract class Decorator : IWindowObject
        {
                protected IWindowObject _decoratedWindow = null; // the object being
decorated

                public Decorator( IWindowObject decoratedWindow )
                {
                        _decoratedWindow = decoratedWindow;
                }

                public virtual void draw()
                {
                        _decoratedWindow.draw();
                        Console.WriteLine("\tDecorator::draw() ");
                }

                public virtual String getDescription() // returns a description of the
object
                {
                        return _decoratedWindow.getDescription() + "\n\t" +
"Decorator::getDescription() ";
                }
        }

        // the first decorator
        class DecorationA : Decorator
        {
                public DecorationA(IWindowObject decoratedWindow) : base(decoratedWindow)
                {
                }

                public override void draw()
                {
                        base.draw();
                        DecorationAStuff();
                }

                private void DecorationAStuff()
                {
                        Console.WriteLine("\t\tdoing DecorationA things");
                }

                public override String getDescription()
                {
                        return base.getDescription() + "\n\t\tincluding " +
this.ToString();
                }

        }// end  class ConcreteDecoratorA : Decorator

        class DecorationB : Decorator
        {
                public DecorationB(IWindowObject decoratedWindow) : base(decoratedWindow)
                {
                }

                public override void draw()
                {
                        base.draw();
                        DecorationBStuff();
                }

                private void DecorationBStuff()
                {
                        Console.WriteLine("\t\tdoing DecorationB things");
                }

                public override String getDescription()
```

```
                {
                        return base.getDescription() + "\n\t\tincluding " +
this.ToString();
                }

        }// end  class DecorationB : Decorator

        class DecorationC : Decorator
        {
                public DecorationC(IWindowObject decoratedWindow) : base(decoratedWindow)
                {
                }

                public override void draw()
                {
                        base.draw();
                        DecorationCStuff();
                }

                private void DecorationCStuff()
                {
                        Console.WriteLine("\t\tdoing DecorationC things");
                }

                public override String getDescription()
                {
                        return base.getDescription() + "\n\t\tincluding " +
this.ToString();
                }

        }// end  class DecorationC : Decorator

}// end of namespace GSL_Decorator_pattern
```

The decorator pattern can also be implemented in dynamic languages with no interfaces or traditional OOP inheritance.

JavaScript coffee shop:

```
//Class to be decorated
function Coffee(){
    this.cost = function(){
        return 1;
    };
}

//Decorator A
function Milk(coffee){
    this.cost = function(){
        return coffee.cost() + 0.5;
    };
}

//Decorator B
function Whip(coffee){
    this.cost = function(){
        return coffee.cost() + 0.7;
    };
}

//Decorator C
function Sprinkles(coffee){
    this.cost = function(){
        return coffee.cost() + 0.2;
    };
}

//Here's one way of using it
```

```
var coffee = new Milk(new Whip(new Sprinkles(new Coffee())));
alert( coffee.cost() );

//Here's another
var coffee = new Coffee();
coffee = new Sprinkles(coffee);
coffee = new Whip(coffee);
coffee = new Milk(coffee);
alert(coffee.cost());
```

Facade pattern

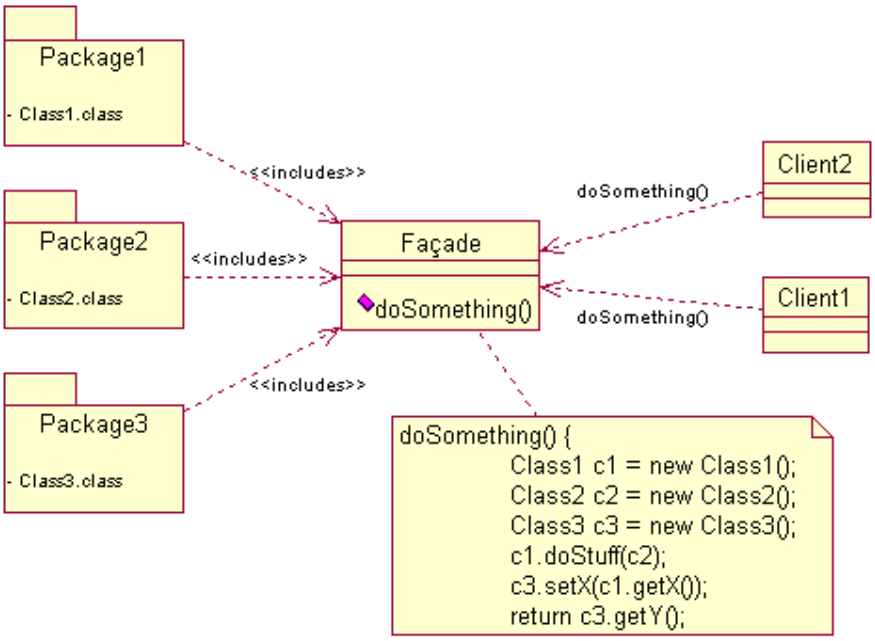
The **facade pattern** or **façade pattern** is a software engineering design pattern commonly used with Object-oriented programming.

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:

> make a software library easier to use and understand, since the facade has convenient methods for common tasks;
> make code that uses the library more readable, for the same reason;
> reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system;
> wrap a poorly-designed collection of APIs with a single well-designed API (as per task needs).

An Adapter is used when the wrapper must respect a particular interface and must support a polymorphic behavior. On the other hand, a facade is used when one wants an easier or simpler interface to work with.

Structure



Facade
: The facade class interacts Packages 1, 2, and 3 with the rest of the application.

Clients
: The objects using the Facade Pattern to access resources from the Packages.

Packages
: Software library / API collection accessed through the Facade Class.

Java

This is an abstract example of how a client ("you") interacts with a façade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive).

```
/* Complex parts */

class CPU {
        public void freeze() { ... }
        public void jump(long position) { ... }
        public void execute() { ... }
}

class Memory {
        public void load(long position, byte[] data) {
                ...
        }
}

class HardDrive {
        public byte[] read(long lba, int size) {
                ...
        }
}

/* Façade */
```

```
class Computer {
        public void startComputer() {
                cpu.freeze();
                memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
                cpu.jump(BOOT_ADDRESS);
                cpu.execute();
        }
}

/* Client */

class You {
        public static void main(String[] args) throws ParseException {
                Computer facade = /* grab a facade instance */;
                facade.startComputer();
        }
}
```

C#

```
// Facade pattern -- Structural example

using System;

namespace DoFactory.GangOfFour.Facade.Structural
{

  // Mainapp test application

  class MainApp
  {
    public static void Main()
    {
      Facade facade = new Facade();

      facade.MethodA();
      facade.MethodB();

      // Wait for user
      Console.Read();
    }
  }

  // "Subsystem ClassA"

  class SubSystemOne
  {
    public void MethodOne()
    {
      Console.WriteLine(" SubSystemOne Method");
    }
  }

  // Subsystem ClassB"

  class SubSystemTwo
  {
    public void MethodTwo()
    {
      Console.WriteLine(" SubSystemTwo Method");
    }
  }

  // Subsystem ClassC"

  class SubSystemThree
  {
    public void MethodThree()
    {
      Console.WriteLine(" SubSystemThree Method");
```

```
    }
  }

  // Subsystem ClassD"

  class SubSystemFour
  {
    public void MethodFour()
    {
      Console.WriteLine(" SubSystemFour Method");
    }
  }

  // "Facade"

  class Facade
  {
    SubSystemOne one;
    SubSystemTwo two;
    SubSystemThree three;
    SubSystemFour four;

    public Facade()
    {
      one = new SubSystemOne();
      two = new SubSystemTwo();
      three = new SubSystemThree();
      four = new SubSystemFour();
    }

    public void MethodA()
    {
      Console.WriteLine("\nMethodA() ---- ");
      one.MethodOne();
      two.MethodTwo();
      four.MethodFour();
    }

    public void MethodB()
    {
      Console.WriteLine("\nMethodB() ---- ");
      two.MethodTwo();
      three.MethodThree();
    }
  }
}
```

Flyweight pattern

Flyweight is a software design pattern. A Flyweight is an object that minimizes memory occupation by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple representation would use an unacceptable amount of memory. Often some parts of the object state can be shared and it's common to put them in external data structures and pass them to the flyweight objects temporarily when they are used.

A classic example usage of the flyweight pattern are the data structures for graphical representation of characters in a word processor. It would be nice to have, for each character in a document, a glyph object containing its font outline, font metrics, and other formatting data, but it would amount to hundreds or thousands of bytes for each character. Instead, for every character there might be a reference to a flyweight glyph

object shared by every instance of the same character in the document; only the position of each character (in the document and/or the page) would need to be stored externally.

The following programs illustrate the document example given above: the flyweights are called FontData in the Java example and GraphicChar in the C# example.

The examples illustrate the Flyweight pattern used to reduce memory by loading only the data necessary to perform some immediate task from a large Font object into a much smaller FontData (Flyweight) object.

Java

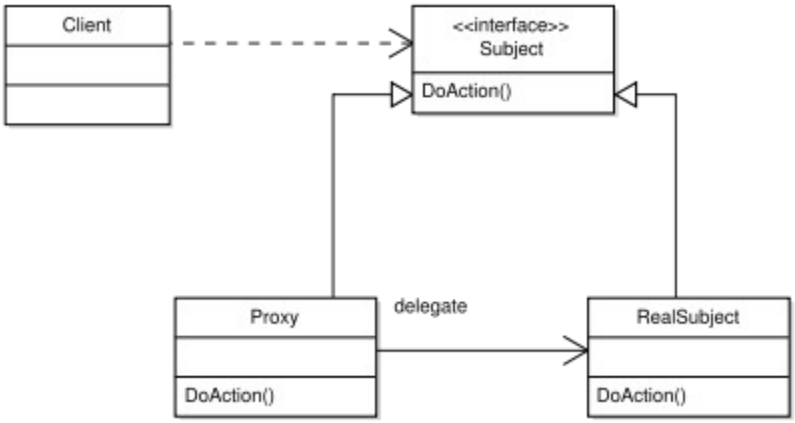
```
public enum FontEffect {
    BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}

public final class FontData {
    /**
     * A weak hash map will drop unused references to FontData.
     * Values have to be wrapped in WeakReferences,
     * because value objects in weak hash map are held by strong references.
     */
    private static final WeakHashMap<FontData, WeakReference<FontData>> flyweightData =
        new WeakHashMap<FontData, WeakReference<FontData>>();
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;

    private FontData(int pointSize, String fontFace, Color color, EnumSet<FontEffect>
effects) {
        this.pointSize = pointSize;
        this.fontFace = fontFace;
        this.color = color;
        this.effects = Collections.unmodifiableSet(effects);
    }

    public static FontData create(int pointSize, String fontFace, Color color,
        FontEffect... effects) {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.class);
        for (FontEffect fontEffect : effects) {
            effectsSet.add(fontEffect);
        }
        // We are unconcerned with object creation cost, we are reducing overall memory
consumption
        FontData data = new FontData(pointSize, fontFace, color, effectsSet);
        if (!flyweightData.containsKey(data)) {
            flyweightData.put(data, new WeakReference(data));
        }
        // return the single immutable copy with the given values
        return flyweightData.get(data).get();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof FontData) {
            if (obj == this) {
                return true;
            }
            FontData other = (FontData) obj;
            return other.pointSize == pointSize && other.fontFace.equals(fontFace)
                && other.color.equals(color) && other.effects.equals(effects);
        }
        return false;
    }
```

```
    @Override
    public int hashCode() {
        return (pointSize * 37 + effects.hashCode() * 13) * fontFace.hashCode();
    }

    // Getters for the font data, but no setters. FontData is immutable.
}
```

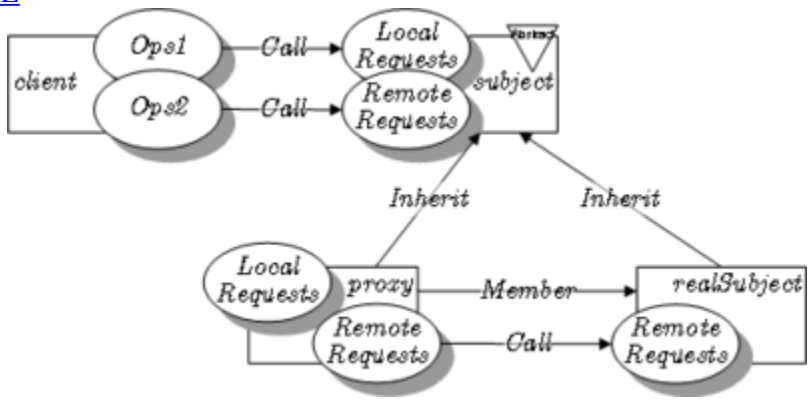
C#

```
using System.Collections;
using System.Collections.Generic;
using System;

class GraphicChar {
char c;
string fontFace;
public GraphicChar(char c, string fontFace) { this.c = c; this.fontFace = fontFace; }
public static void printAtPosition(GraphicChar c, int x, int y)   {
Console.WriteLine("Printing '{0}' in '{1}' at position {2}:{3}.", c.c, c.fontFace, x, y);
}
}

class GraphicCharFactory {
Hashtable pool = new Hashtable(); // the Flyweights

public int getNum() { return pool.Count; }

public GraphicChar get(char c, string fontFace) {
GraphicChar gc;
string key = c.ToString() + fontFace;
gc = pool[key] as GraphicChar;
if (gc == null) {
gc = new GraphicChar(c, fontFace);
pool.Add(key, gc);
}
return gc;
}
}

class FlyWeightExample {
public static void Main(string[] args) {
GraphicCharFactory cf = new GraphicCharFactory();

// Compose the text by storing the characters as objects.
List<GraphicChar> text = new List<GraphicChar>();
text.Add(cf.get('H', "Arial"));    // 'H' and "Arial" are called intrinsic information
text.Add(cf.get('e', "Arial"));    // because it is stored in the object itself.
text.Add(cf.get('l', "Arial"));
text.Add(cf.get('l', "Arial"));
text.Add(cf.get('o', "Times"));

// See how the Flyweight approach is beginning to save space:
Console.WriteLine("CharFactory created only {0} objects for {1} characters.",
cf.getNum(), text.Count);

int x=0, y=0;
foreach (GraphicChar c in text) {             // Passing position as extrinsic
information to the objects,
GraphicChar.printAtPosition(c, x++, y);   // as a top-left 'A' is not different from a
top-right one.
}
}
}
```

Proxy pattern



Proxy in UML



Proxy in LePUS3 (legend)

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

A proxy, in its most general form, is a class functioning as an interface to another thing. The other thing could be anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.

A well-known example of the proxy pattern is a reference counting pointer object.

In situations where multiple copies of a complex object must exist the proxy pattern can be adapted to incorporate the Flyweight Pattern in order to reduce the application's memory footprint. Typically one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory may be deallocated.

Virtual proxy (in Java)

The following Java example illustrates the "virtual proxy" pattern. The program's output is:

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

The `ProxyImage` class is used to delay the expensive operation of loading a file from disk until the result of that operation is actually needed. If the file is never needed, then the expensive load has been totally eliminated.

```
import java.util.*;

interface Image {
    public void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        // Potentially expensive operation
        // ...
        System.out.println("Loading   "+filename);
    }

    public void displayImage() { System.out.println("Displaying "+filename); }
}

class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) { this.filename = filename; }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); // load only on demand
        }
        image.displayImage();
    }
}

class ProxyExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("HiRes_10MB_Photo1");
        Image image2 = new ProxyImage("HiRes_10MB_Photo2");
        Image image3 = new ProxyImage("HiRes_10MB_Photo3");

        image1.displayImage(); // loading necessary
        image2.displayImage(); // loading necessary
        image2.displayImage(); // no loading necessary; already done
        // the third image will never be loaded - time saved!
    }
}
```

Protection proxy (in C#)

In this C# example, the `RealClient` stores an account number. Only users who know a valid password can access this account number. The `RealClient` is protected by a `ProtectionProxy` which knows the password. If a user wants to get an account number, first the proxy asks the user to authenticate; only if the user entered a correct password does the proxy invoke the `RealClient` to get an account number for the user.

In this example, **thePassword** is the correct password.

```
using System;

namespace ConsoleApplicationTest.FundamentalPatterns.ProtectionProxyPattern
{
    public interface IClient {
        string GetAccountNo();
    }

    public class RealClient : IClient {
        private string accountNo = "12345";
        public RealClient() {
            Console.WriteLine("RealClient: Initialized");
        }
        public string GetAccountNo() {
            Console.WriteLine("RealClient's AccountNo: " + accountNo);
            return accountNo;
        }
    }


    public class ProtectionProxy : IClient
    {
        private string password;  //password to get secret
        RealClient client;

        public ProtectionProxy(string pwd) {
            Console.WriteLine("ProtectionProxy: Initialized");
            password = pwd;
            client = new RealClient();
        }

        // Authenticate the user and return the Account Number
        public String GetAccountNo() {
            Console.Write("Password: ");
            string tmpPwd = Console.ReadLine();

            if (tmpPwd == password) {
                return client.GetAccountNo();
            } else {
                Console.WriteLine("ProtectionProxy: Illegal password!");
                return "";
            }
        }
    }

    class ProtectionProxyExample
    {
        [STAThread]
        public static void Main(string[] args) {
            IClient client = new ProtectionProxy("thePassword");
            Console.WriteLine();
            Console.WriteLine("main received: " + client.GetAccountNo());
            Console.WriteLine("\nPress any key to continue . . .");
            Console.Read();
        }
```

```
    }
}
```

Chain-of-responsibility pattern

From Wikipedia, the free encyclopedia

Jump to: navigation, search

In Object Oriented Design, the **chain-of-responsibility pattern** is a design pattern consisting of a source of command objects and a series of **processing objects**. Each processing object contains a set of logic that describes the types of command objects that it can handle, and how to pass off those that it cannot to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

In a variation of the standard chain-of-responsibility model, some handlers may act as dispatchers, capable of sending commands out in a variety of directions, forming a *tree of responsibility*. In some cases, this can occur recursively, with processing objects calling higher-up processing objects with commands that attempt to solve some smaller part of the problem; in this case recursion continues until the command is processed, or the entire tree has been explored. An XML interpreter (parsed, but not yet executed) might be a fitting example.

This pattern promotes the idea of loose coupling, which is considered a programming best practice.

Java

The following Java code illustrates the pattern with the example of a logging class. Each logging handler decides if any action is to be taken at this log level and then passes the message on to the next logging handler. The output is:

```
Writing to output:   Entering function y.
Writing to output:   Step1 completed.
Sending via e-mail:  Step1 completed.
Writing to output:   An error has occurred.
Sending via e-mail:  An error has occurred.
Writing to stderr:   An error has occurred.
```

Note that this example should not be seen as a recommendation to write Logging classes this way.

Also, note that in a 'pure' implementation of the CoR a logger would not pass responsibility further down the chain after handling a message. In this example a message will be passed down the chain whether it is handled or not.

```
import java.util.*;
```

```
abstract class Logger
{
    public static int ERR = 3;
    public static int NOTICE = 5;
    public static int DEBUG = 7;
    protected int mask;

    // The next element in the chain of responsibility
    protected Logger next;
    public Logger setNext( Logger l)
    {
        next = l;
        return l;
    }

    public void message( String msg, int priority )
    {
        if ( priority <= mask )
        {
            writeMessage( msg );
            if ( next != null )
            {
                next.message( msg, priority );
            }
        }
    }

    abstract protected void writeMessage( String msg );

}

class StdoutLogger extends Logger
{

    public StdoutLogger( int mask ) { this.mask = mask; }

    protected void writeMessage( String msg )
    {
        System.out.println( "Writing to stdout: " + msg );
    }
}


class EmailLogger extends Logger
{

    public EmailLogger( int mask ) { this.mask = mask; }

    protected void writeMessage( String msg )
    {
        System.out.println( "Sending via email: " + msg );
    }
}

class StderrLogger extends Logger
{

    public StderrLogger( int mask ) { this.mask = mask; }

    protected void writeMessage( String msg )
    {
        System.out.println( "Sending to stderr: " + msg );
    }
}

public class ChainOfResponsibilityExample
{
    public static void main( String[] args )
    {
        // Build the chain of responsibility
```

```
        Logger l,l1;
        l1 = l = new StdoutLogger( Logger.DEBUG );
        l1 = l1.setNext(new EmailLogger( Logger.NOTICE ));
        l1 = l1.setNext(new StderrLogger( Logger.ERR ));

        // Handled by StdoutLogger
        l.message( "Entering function y.", Logger.DEBUG );

        // Handled by StdoutLogger and EmailLogger
        l.message( "Step1 completed.", Logger.NOTICE );

        // Handled by all three loggers
        l.message( "An error has occurred.", Logger.ERR );
    }
}
```

C#, Abstract class based implementation

```
using System;

namespace Chain_of_responsibility
{
        public abstract class Chain
        {
                private Chain _next;

                public Chain Next
                {
                        get
                        {
                                return _next;
                        }
                        set
                        {
                                _next = value;
                        }
                }

                public void Message(object command)
                {
                        if ( Process(command) == false && _next != null )
                        {
                                _next.Message(command);
                        }
                }

                public static Chain operator +(Chain lhs, Chain rhs)
                {
                        Chain last = lhs;

                        while ( last.Next != null )
                        {
                                last = last.Next;
                        }

                        last.Next = rhs;

                        return lhs;
                }

                protected abstract bool Process(object command);
        }

        public class StringHandler : Chain
        {
                protected override bool Process(object command)
                {
                        if ( command is string )
                        {
                                Console.WriteLine("StringHandler can handle this message
: {0}",(string)command);
```

```
                        return true;
                    }

                    return false;
                }
        }

        public class IntegerHandler : Chain
        {
                protected override bool Process(object command)
                {
                        if ( command is int )
                        {
                                Console.WriteLine("IntegerHandler can handle this message
: {0}",(int)command);

                                return true;
                        }

                        return false;
                }
        }

        public class NullHandler : Chain
        {
                protected override bool Process(object command)
                {
                        if ( command == null )
                        {
                                Console.WriteLine("NullHandler can handle this message.");

                                return true;
                        }

                        return false;
                }
        }

        public class IntegerBypassHandler : Chain
        {
                protected override bool Process(object command)
                {
                        if ( command is int )
                        {
                                Console.WriteLine("IntegerBypassHandler can handle this
message : {0}",(int)command);

                                return false; // Always pass to next handler
                        }

                        return false; // Always pass to next handler
                }
        }

        class TestMain
        {
                static void Main(string[] args)
                {
                        Chain chain = new StringHandler();
                        chain += new IntegerBypassHandler();
                        chain += new IntegerHandler();
                        chain += new IntegerHandler(); // Never reached
                        chain += new NullHandler();

                        chain.Message("1st string value");
                        chain.Message(100);
                        chain.Message("2nd string value");
                        chain.Message(4.7f); // not handled
                        chain.Message(null);
                }
```

```
        }
}
```

Interface based implementation

```
using System;
using System.Collections;

namespace Chain_of_responsibility
{
        public interface IChain
        {
                bool Process(object command);
        }

        public class Chain
        {
                private ArrayList _list;

                public ArrayList List
                {
                        get
                        {
                                return _list;
                        }
                }

                public Chain()
                {
                        _list = new ArrayList();
                }

                public void Message(object command)
                {
                        foreach ( IChain item in _list )
                        {
                                bool result = item.Process(command);

                                if ( result == true ) break;
                        }
                }

                public void Add(IChain handler)
                {
                        List.Add(handler);
                }
        }

        public class StringHandler : IChain
        {
                public bool Process(object command)
                {
                        if ( command is string )
                        {
                                Console.WriteLine("StringHandler can handle this message
: {0}",(string)command);

                                return true;
                        }

                        return false;
                }
        }

        public class IntegerHandler : IChain
        {
                public bool Process(object command)
                {
                        if ( command is int )
                        {
```

```
                                Console.WriteLine("IntegerHandler can handle this message
: {0}",(int)command);

                                return true;
                        }

                        return false;
                }
        }

        public class NullHandler : IChain
        {
                public bool Process(object command)
                {
                        if ( command == null )
                        {
                                Console.WriteLine("NullHandler can handle this message.");

                                return true;
                        }

                        return false;
                }
        }

        public class IntegerBypassHandler : IChain
        {
                public bool Process(object command)
                {
                        if ( command is int )
                        {
                                Console.WriteLine("IntegerBypassHandler can handle this
message : {0}",(int)command);

                                return false; // Always pass to next handler
                        }

                        return false; // Always pass to next handler
                }
        }

        class TestMain
        {
                static void Main(string[] args)
                {
                        Chain chain = new Chain();

                        chain.Add(new StringHandler());
                        chain.Add(new IntegerBypassHandler());
                        chain.Add(new IntegerHandler());
                        chain.Add(new IntegerHandler()); // Never reached
                        chain.Add(new NullHandler());

                        chain.Message("1st string value");
                        chain.Message(100);
                        chain.Message("2nd string value");
                        chain.Message(4.7f); // not handled
                        chain.Message(null);
                }
        }
}
```

Delegate based implementation

```
using System;

namespace Chain_of_responsibility
{
        public class StringHandler
        {
                private static int count;
```

```
                public void Process(object command)
                {
                        if ( command is string )
                        {
                                string th;
                                count++;

                                if ( count % 10 == 1 ) th = "st";
                                else if ( count % 10 == 2 ) th = "nd";
                                else if ( count % 10 == 3 ) th = "rd";
                                else th = "th";

                                Console.WriteLine("StringHandler can handle this {0}{1}
message : {2}",count,th,(string)command);
                        }
                }
        }

        public class StringHandler2
        {
                public void Process(object command)
                {
                        if ( command is string )
                        {
                                Console.WriteLine("StringHandler2 can handle this message
: {0}",(string)command);
                        }
                }
        }

        public class IntegerHandler
        {
                public void Process(object command)
                {
                        if ( command is int )
                        {
                                Console.WriteLine("IntegerHandler can handle this message
: {0}",(int)command);
                        }
                }
        }

        class Chain
        {
                public delegate void MessageHandler(object message);
                public static event MessageHandler Message;

                public static void Process(object message)
                {
                        Message(message);
                }
        }

        class TestMain
        {
                static void Main(string[] args)
                {
                        StringHandler sh1 = new StringHandler();
                        StringHandler2 sh2 = new StringHandler2();
                        IntegerHandler ih = new IntegerHandler();

                        Chain.Message += new Chain.MessageHandler(sh1.Process);
                        Chain.Message += new Chain.MessageHandler(sh2.Process);
                        Chain.Message += new Chain.MessageHandler(ih.Process);
                        Chain.Message += new Chain.MessageHandler(ih.Process); // Can
also be reached

                        Chain.Process("1st string value");
                        Chain.Process(100);
                        Chain.Process("2nd string value");
```

```
                        Chain.Process(4.7f); // not handled
                }
        }
}
```

PHP

```
<?php
abstract class Logger {
        const ERR = 3;
        const NOTICE = 5;
        const DEBUG = 7;

        protected $mask;
        protected $next; // The next element in the chain of responsibility

        public function setNext(Logger $l) {
                $this->next = $l;
                return $this;
        }

        public function message($msg, $priority) {

                if ($priority <= $this->mask) {
                        $this->writeMessage( $msg );
                }

                if (false == is_null($this->next)) {
                        $this->next->message($msg, $priority);
                }
        }

        abstract public function writeMessage($msg );
}

class DebugLogger extends Logger {
        public function __construct($mask) {
                $this->mask = $mask;
                return $this;
        }

        public function writeMessage($msg ) {
                        echo "Writing to debug output: {$msg}\n";
        }
}

class EmailLogger extends Logger {
        public function __construct($mask) {
                $this->mask = $mask;
                return $this;
        }

        public function writeMessage($msg ) {
                        echo "Sending via email: {$msg}\n";
        }
}

class StderrLogger extends Logger {
        public function __construct($mask) {
                $this->mask = $mask;
                return $this;
        }

        public function writeMessage($msg ) {
                        echo "Writing to stderr: {$msg}\n";
        }
}

class ChainOfResponsibilityExample {
        public function __construct() {
                // build the chain of responsibility
```

```
                $l = new DebugLogger(Logger::DEBUG);
                $e = new EmailLogger(Logger::NOTICE);
                $s = new StderrLogger(Logger::ERR);
                $l->setNext( $e->setNext( $s ) );

                $l->message("Entering function y.",                Logger::DEBUG);
// handled by DebugLogger
                $l->message("Step1 completed.",                    Logger::NOTICE);  //
handled by DebugLogger and EmailLogger
                $l->message("An error has occurred.",      Logger::ERR);             //
handled by all three Loggers
        }
}

new ChainOfResponsibilityExample();
```

Visual Prolog

This is a realistic example of protecting the operations on a stream with a named mutex.

First the outputStream interface (a simplified version of the real one):

```
interface outputStream
   predicates
      write : (...).
      writef : (string FormatString, ...).
end interface outputStream
```

This class encapsulates each of the stream operations the mutex. The **finally** predicate is used to ensure that the mutex is released no matter how the operation goes (i.e. also in case of exceptions). The underlying mutex object is released by a *finalizer* in the mutex class, and for this example we leave it at that.

```
class outputStream_protected : outputStream
   constructors
      new : (string MutexName, outputStream Stream).
end class outputStream_protected

#include @"pfc\multiThread\multiThread.ph"

implement outputStream_protected
   facts
      mutex : mutex.
      stream : outputStream.
   clauses
      new(MutexName, Stream) :-
         mutex := mutex::createNamed(MutexName, false), % do not take ownership
         stream := Stream.
   clauses
      write(...) :-
         _ = mutex:wait(),  % ignore wait code in this simplified example
         finally(stream:write(...), mutex:release()).

   clauses
      writef(FormatString, ...) :-
         _ = mutex:wait(),  % ignore wait code in this simplified example
         finally(stream:writef(FormatString, ...), mutex:release()).
end implement outputStream_protected
```

Usage example.

The **client** uses an **outputStream**. Instead of receiving the **pipeStream** directly, it gets the protected version of it.

```
#include @"pfc\pipe\pipe.ph"

goal
   Stream = pipeStream::openClient("TestPipe"),
   Protected = outputStream_protected::new("PipeMutex", Stream),
   client(Protected),
   Stream:close().
```

ColdFusion

```
<cfcomponent name="AbstractLogger" hint="Base Logging Class">

        <cfset this['ERROR'] = 3 />
        <cfset this['NOTICE'] = 5 />
        <cfset this['DEBUG'] = 7 />
        <cfset mask = 0 />

        <cffunction name="init" access="public" returntype="AbstractLogger"
hint="Constructor.">
                <cfreturn this />
        </cffunction>

        <cffunction name="setMask" access="public" returntype="AbstractLogger"
output="false" hint="I set the mask value.">
                <cfargument name="maskName" type="string" required="true" />
                <cfset mask = this[arguments.maskName] />
                <cfreturn this />
        </cffunction>

        <cffunction name="setNext" access="public" returntype="AbstractLogger"
output="false" hint="I set the next Logger in the chain">
                <cfargument name="logger" type="AbstractLogger" required="true" />
                <cfset next = arguments.logger />
                <cfreturn this />
        </cffunction>

        <cffunction name="message" access="public" returntype="void" output="true"
hint="I generate the Log Message.">
                <cfargument name="msg" type="string" required="true" />
                <cfargument name="priority" type="numeric" required="true" />
                <cfset var local = StructNew() />
                <cfif arguments.priority lte mask>
                        <cfset writeMessage(arguments.msg) />
                        <cfif StructKeyExists(variables, 'next')>
                                <cfset next.message(arguments.msg, arguments.priority) />
                        </cfif>
                </cfif>
        </cffunction>

</cfcomponent>

<cfcomponent name="DebugLogger" extends="AbstractLogger" hint="Standard Logger">

        <cffunction name="init" access="public" returntype="DebugLogger"
hint="Constructor.">
                <cfreturn this />
        </cffunction>

        <cffunction name="writeMessage" access="public" returntype="void" output="true"
hint="Generate the message.">
                <cfargument name="msg" type="string" required="true" />
```

```
                <cfoutput>Writing to debug: #arguments.msg#</cfoutput>
        </cffunction>

</cfcomponent>

<cfcomponent name="EmailLogger" extends="AbstractLogger" hint="Email Logger">

        <cffunction name="init" access="public" returntype="EmailLogger"
hint="Constructor.">
                <cfreturn this />
        </cffunction>

        <cffunction name="writeMessage" access="public" returntype="void" output="true"
hint="Generate the message.">
                <cfargument name="msg" type="string" required="true" />
                <cfoutput>Sending via email: #arguments.msg#</cfoutput>
        </cffunction>

</cfcomponent>

<cfcomponent name="ErrorLogger" extends="AbstractLogger" hint="Error Logger">

        <cffunction name="init" access="public" returntype="ErrorLogger"
hint="Constructor.">
                <cfreturn this />
        </cffunction>

        <cffunction name="writeMessage" access="public" returntype="void" output="true"
hint="Generate the message.">
                <cfargument name="msg" type="string" required="true" />
                <cfoutput>Sending to error: #arguments.msg#</cfoutput>
        </cffunction>

</cfcomponent>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html>
<head>
<title>ColdFusion Example of Chain of Responsibility Pattern</title>
</head>

<body>
<h1>ColdFusion Example of Chain of Responsibility Pattern</h1>

<cfset logger = CreateObject('component', 'DebugLogger').init().setMask('DEBUG') />
<cfset emailLogger = CreateObject('component', 'EmailLogger').init().setMask('NOTICE') />
<cfset errorLogger = CreateObject('component', 'ErrorLogger').init().setMask('ERROR') />
<cfset logger.setNext(emailLogger.setNext(errorLogger)) />

<!--- Handled by StdoutLogger --->
<cfset logger.message("Entering function y.", logger['DEBUG']) /><br /><br />
<!--- Handled by StdoutLogger and EmailLogger --->
<cfset logger.message("Step1 completed.", logger['NOTICE']) /><br /><br />
<!--- Handled by all three loggers --->
<cfset logger.message("An error has occurred.", logger['ERROR']) /><br /><br />

</body>
</html>
```

Command pattern

From Wikipedia, the free encyclopedia

Jump to: navigation, search

In object-oriented programming, the **Command pattern** is a design pattern in which objects are used to represent actions. A **command object** encapsulates an action and its parameters.

For example, a printing library might include a `PrintJob` class. A user would typically create a new `PrintJob` object, set its properties (the document to be printed, the number of copies, and so on), and finally call a method to send the job to the printer.

In this case, the same functionality could be exposed via a single `SendJobToPrinter()` procedure with many parameters. As it takes more code to write a command class than to write a procedure, there is often a reason to use a class. There are many possible reasons:

> A command object is convenient temporary storage for procedure parameters. It can be used while assembling the parameters for a function call and allows the command to be set aside for later use.
> A class is a convenient place to collect code and data related to a command. A command object can hold information about the command, such as its name or which user launched it; and answer questions about it, such as how long it will likely take. It also allows the command to be executed some time after it is defined.
> Treating commands as objects enables data structures containing multiple commands. A complex process could be treated as a tree or graph of command objects. A thread pool could maintain a priority queue of command objects consumed by worker threads.
> Treating commands as objects supports undo-able operations, provided that the command objects are stored (for example in a stack).
> The command is a useful abstraction for building generic components, such as a thread pool, that can handle command objects of any type. If a new type of command object is created later, it can work with these generic components automatically. For example, in Java, a generic `ThreadPool` class could have a method `addTask(Runnable task)` that accepts any object that implements the `java.lang.Runnable` interface(see ThreadPoolExecutor.execute(Runnable task)).

Uses for the Command pattern

Command objects are useful for implementing:

Multi-level undo
> If all user actions in a program are implemented as command objects, the program can keep a stack of the most recently executed commands. When the user wants to undo a command, the program simply pops the most recent command object and executes its `undo()` method.

Transactional behavior
> Undo is perhaps even more essential when it's called *rollback* and happens automatically when an operation fails partway through. Installers need this and so

do databases. Command objects can also be used to implement two-phase commit.

Progress bars
: Suppose a program has a sequence of commands that it executes in order. If each command object has a `getEstimatedDuration()` method, the program can easily estimate the total duration. It can show a progress bar that meaningfully reflects how close the program is to completing all the tasks.

Wizards
: Often a wizard presents several pages of configuration for a single action that happens only when the user clicks the "Finish" button on the last page. In these cases, a natural way to separate user interface code from application code is to implement the wizard using a command object. The command object is created when the wizard is first displayed. Each wizard page stores its GUI changes in the command object, so the object is populated as the user progresses. "Finish" simply triggers a call to `execute()`. This way, the command class contains no user interface code.

GUI buttons and menu items
: In Swing and Borland Delphi programming, an `Action` is a command object. In addition to the ability to perform the desired command, an `Action` may have an associated icon, keyboard shortcut, tooltip text, and so on. A toolbar button or menu item component may be completely initialized using only the `Action` object.

Thread pools
: A typical, general-purpose thread pool class might have a public `addTask()` method that adds a work item to an internal queue of tasks waiting to be done. It maintains a pool of threads that execute commands from the queue. The items in the queue are command objects. Typically these objects implement a common interface such as `java.lang.Runnable` that allows the thread pool to execute the command even though the thread pool class itself was written without any knowledge of the specific tasks for which it would be used.

Macro recording
: If all user actions are represented by command objects, a program can record a sequence of actions simply by keeping a list of the command objects as they are executed. It can then "play back" the same actions by executing the same command objects again in sequence. If the program embeds a scripting engine, each command object can implement a `toScript()` method, and user actions can then be easily recorded as scripts.

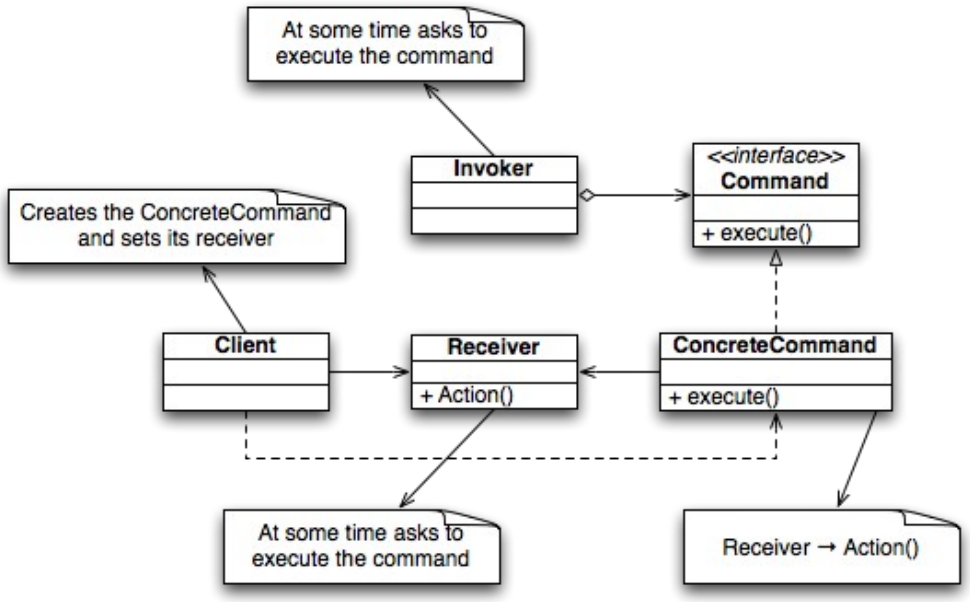
Networking
: It is possible to send whole command objects across the network to be executed on the other machines, for example player actions in computer games.

Parallel Processing
: Where the commands are written as tasks to a shared resource and executed by many threads in parallel (possibly on remote machines -this variant is often referred to as the Master/Worker pattern)

Mobile Code

Using languages such as Java where code can be streamed/slurped from one location to another via URLClassloaders and Codebases the commands can enable new behavior to be delivered to remote locations (EJB Command, Master Worker)

Structure



Terminology

The terminology used to describe command pattern implementations is not consistent and can therefore be confusing.

Client, Source: the button, toolbar button, or menu item clicked, the shortcut key pressed by the user.
Invoker, Command Object, Routed Command Object, Action Object: a singleton object (e.g. there is only one CopyCommand object), which knows about shortcut keys, button images, command text, etc. related to the command. A client/source object calls the Command/Action object's execute/performAction method. The Command/Action object notifies the appropriate client/source objects when the availability of a command/action has changed. This allows buttons and menu items to become inactive (grayed out) when a command/action cannot be executed/performed.
Receiver, Target Object: the object that is about to be copied, pasted, moved, etc.
Command Object, routed event args, event object: the object that is passed from the source to the Command/Action object, to the Target object to the code that does the work. Each button click or shortcut key results in a new command/event object. Some implementations add more information to the

command/event object as it is being passed from one object (e.g. CopyCommand) to another (e.g. document section). Other implementations put command/event objects in other event objects (like a box inside a bigger box) as they move along the line, to avoid naming conflicts.

Handler, ExecutedRoutedEventHandler, method, function: the actual code that does the copying, pasting, moving, etc. In some implementations the handler code is part of the command/action object. In other implementations the code is part of the Receiver/Target Object, and in yet other implementations the handler code is kept separate from the other objects.

Command Manager, Undo Manager, Scheduler, Queue, Dispatcher, Invoker: an object that puts command/event objects on an undo stack or redo stack, or that holds on to command/event objects until other objects are ready to act on them, or that routes the command/event objects to the appropriate receiver/target object or handler code.

Example (C++)

Consider a simple recipe making program. Every command could be represented as an object. Then if the user makes a mistake then he/she can undo the command by removing that object from the stack

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

class Command{
  public:
   virtual void execute(void) =0;
   virtual ~Command(void){};
};

class Ingredient : public Command {
  public:
   Ingredient(string amount, string ingredient){
      _ingredient = ingredient;
      _amount = amount;
   }
   void execute(void){
      cout << " *Add " << _amount << " of " << _ingredient << endl;
   }
  private:
   string _ingredient;
   string _amount;
};

class Step : public Command {
  public:
   Step(string action, string time){
      _action= action;
      _time= time;
   }
   void execute(void){
      cout << " *" << _action << " for " << _time << endl;
   }
  private:
   string _time;
   string _action;
```

```
};

class CmdStack{
  public:
   void add(Command *c) {
      commands.push_back(c);
   }
   void createRecipe(void){
      for(vector<Command*>::size_type x=0;x<commands.size();x++){
         commands[x]->execute();
      }
   }
   void undo(void){
      if(commands.size() > 1) {
         commands.pop_back();
      }
      else {
         cout << "Can't undo" << endl;
      }
   }
  private:
   vector<Command*> commands;
};

int main(void) {
   CmdStack list;

   //Create ingredients
   Ingredient first("2 tablespoons", "vegetable oil");
   Ingredient second("3 cups", "rice");
   Ingredient third("1 bottle","Ketchup");
   Ingredient fourth("4 ounces", "peas");
   Ingredient fifth("1 teaspoon", "soy sauce");

   //Create Step
   Step step("Stir-fry","3-4 minutes");

   //Create Recipe
   cout << "Recipe for simple Fried Rice" << endl;
   list.add(&first);
   list.add(&second);
   list.add(&step);
   list.add(&third);
   list.undo();
   list.add(&fourth);
   list.add(&fifth);
   list.createRecipe();
   cout << "Enjoy!" << endl;
   return 0;
}
```

Example (Java)

Consider a simple switch. We can use this switch for different devices. The key is to configure the switch to execute some command for flipping it up and for flipping it down. In this example we configure the switch with 2 commands: to turn on the light and to turn off the light. Notice that you can use another Receiver or you can configure the switch dynamically. For example you could configure the switch to open the light and then you can configure it to start an engine. So you can use this switch for different devices.

```
/*the Invoker class*/
public class Switch {
    private Command flipUpCommand;
    private Command flipDownCommand;
```

```
    public Switch(Command flipUpCmd,Command flipDownCmd){
            this.flipUpCommand=flipUpCmd;
            this.flipDownCommand=flipDownCmd;
           }

    public void flipUp(){
         flipUpCommand.execute();
    }

    public void flipDown(){
         flipDownCommand.execute();
    }
}

/*Receiver class*/

public class Light{
     public Light(){  }

     public void turnOn(){
        System.out.println("The light is on");
     }

     public void turnOff(){
        System.out.println("The light is off");
     }
}


/*the Command interface*/

public interface Command{
    void execute();
}


/*the Command for turning on the light*/

public class TurnOnLightCommand implements Command{
   private Light theLight;

   public TurnOnLightCommand(Light light){
        this.theLight=light;
       }

   public void execute(){
      theLight.turnOn();
   }
}

/*the Command for turning off the light*/

public class TurnOffLightCommand implements Command{
   private Light theLight;

   public TurnOffLightCommand(Light light){
        this.theLight=light;
       }

   public void execute(){
      theLight.turnOff();
   }
}

/*The test class*/
public class TestCommand{
   public static void main(String[] args){
       Light l=new Light();
       Command switchUp=new TurnOnLightCommand(l);
       Command switchDown=new TurnOffLightCommand(l);
```

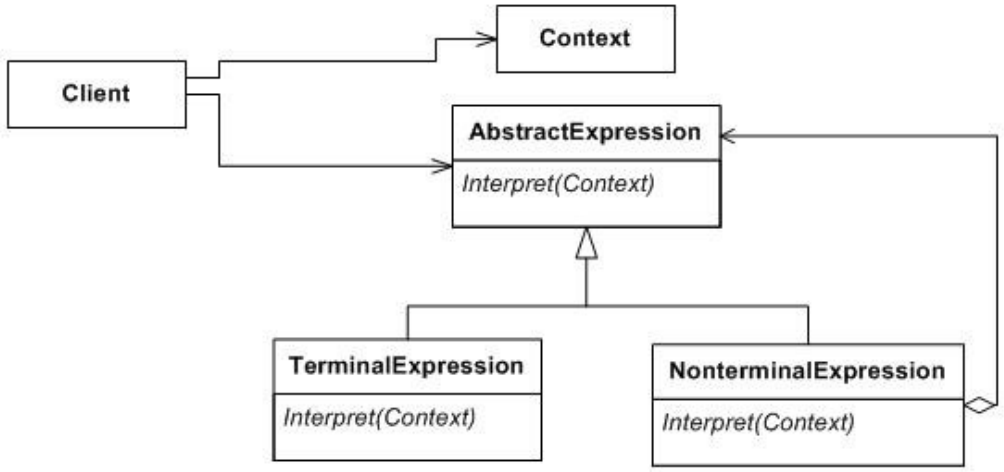
```
        Switch s=new Switch(switchUp,switchDown);

        s.flipUp();
        s.flipDown();
    }
}
```

Interpreter pattern

In computer programming, the **interpreter pattern** is a particular design pattern. The basic idea is to implement a specialized computer language to rapidly solve a defined class of problems. Specialized languages often let a problem be solved several to several hundred times more quickly than a general purpose language would permit.

The general idea is to have a class for each symbol (terminal or nonterminal) in the language.

Structure



Uses for the Interpreter pattern

Specialized database query languages such as SQL.
Specialized computer languages which are often used to describe communication protocols.
Most general-purpose computer languages actually incorporate several specialized languages.

Java

The following Java example illustrates how a general purpose language would interpret a more specialized language, here the Reverse Polish notation. The output is:

```
'42 4 2 - +' equals 44
import java.util.*;

interface Expression {
    public void interpret(Stack<Integer> s);
}

class TerminalExpression_Number implements Expression {
    private int number;
    public TerminalExpression_Number(int number)       { this.number = number; }
    public void interpret(Stack<Integer> s)  { s.push(number); }
}

class TerminalExpression_Plus implements Expression {
    public void interpret(Stack<Integer> s)  { s.push( s.pop() + s.pop() ); }
}

class TerminalExpression_Minus implements Expression {
    public void interpret(Stack<Integer> s)  { s.push( - s.pop() + s.pop() ); }
}

class Parser {
    private ArrayList<Expression> parseTree = new ArrayList<Expression>(); // only one
NonTerminal Expression here

    public Parser(String s) {
        for (String token : s.split(" ")) {
            if      (token.equals("+")) parseTree.add( new TerminalExpression_Plus() );
            else if (token.equals("-")) parseTree.add( new TerminalExpression_Minus() );
            // ...
            else                        parseTree.add( new
TerminalExpression_Number(Integer.valueOf(token)) );
        }
    }

    public int evaluate() {
        Stack<Integer> context = new Stack<Integer>();
        for (Expression e : parseTree) e.interpret(context);
        return context.pop();
    }
}

class InterpreterExample {
    public static void main(String[] args) {
        String expression = "42 4 2 - +";
        Parser p = new Parser(expression);
        System.out.println("'" + expression +"' equals " + p.evaluate());
    }
}
```

Iterator pattern

In object-oriented programming, the **Iterator pattern** is a design pattern in which iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation. An **Iterator object** encapsulates the internal structure of how the iteration occurs.

For example, a tree, linked list, hash table, and an array all need to be iterated with Search, Sort, Next. Rather than having 12 different methods to manage (4 times the previous 3 functions), using this Pattern you would need to manage only 7 ie one for each class using the iterator to obtain the iterator, and one for each of the 3 functions (defined on the iterator itself). Therefore, to run the Search method on the array, you would call array.search, which hides the call to array.iterator.search

PHP 5

As a default behavior in PHP 5, using an object in a foreach structure will traverse all public values. Multiple Iterator classes are available with PHP to allow you to iterate through common lists, such as directories, XML structures and recursive arrays.

It's possible to define your own Iterator classes by implementing the Iterator interface, which will override the default behavior.

The Iterator interface definition:

```
interface Iterator
{
    // Returns the current value
    function current();

    // Returns the current key
    function key();

    // Moves the internal pointer to the next element
    function next();

    // Moves the internal pointer to the first element
    function rewind();

    // If the current element is not at all valid (boolean)
    function valid();
}
```

These methods are all being used in a complete `foreach( $object as $key=>$value )` sequence. The methods are executed in the following order:

```
rewind()
if valid() {
    current() in $value
    key() in $key
    next()
}
End of Loop
```

According to Zend, the `current()` method is called before and after the valid() method

It is also called as : Cursor pattern.

C++

The following C++ program gives the implementation of iterator design pattern with generic C++ template:

Console output:

```
_____Iterator with int_____
0
1
```

```
2
3
4
5
6
7
8
9
_________________Iterator with Class Money_____________________________
100
1000
10000
_________________Set Iterator with Class Name_____________________________
Amt
Bmt
Cmt
Qmt
/************************************************************************/
/* Iterator.h                                                           */
/************************************************************************/
#ifndef MY_ITERATOR_HEADER
#define MY_ITERATOR_HEADER

//////////////////////////////////////////////////////////////////////////
template<class T,class U>
class Iterator
{
public:
        typedef std::vector<T>::iterator iter_type;
        Iterator(U*  pData):m_pData(pData){
                m_it = m_pData->m_data.begin();
        }

        void first()
        {
                m_it = m_pData->m_data.begin();
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }
private:
        U* m_pData;
        iter_type m_it;
};

template<class T,class U,class A>
class setIterator
{
public:
        typedef std::set<T,U>::iterator iter_type;

        setIterator(A* pData):m_pData(pData)
        {
                m_it = m_pData->m_data.begin();
        }

        void first()
        {
                m_it = m_pData->m_data.begin();
```

```
        }

        void next()
        {
                m_it++;
        }

        bool isDone()
        {
                return (m_it == m_pData->m_data.end());
        }

        iter_type current()
        {
                return m_it;
        }

private:
        A*                              m_pData;
        iter_type               m_it;
};
#endif
/*************************************************************************/
/* Aggregate.h                                                           */
/*************************************************************************/
#ifndef MY_DATACOLLECTION_HEADER
#define MY_DATACOLLECTION_HEADER

#include "MyIterator.h"

template <class T>
class aggregate
{
        friend Iterator<T,aggregate>;
public:
        void add(T a)
        {
                m_data.push_back(a);
        }

        Iterator<T,aggregate>* create_iterator()
        {
                return new Iterator<T,aggregate>(this);
        }


private:
        std::vector<T> m_data;
};
template <class T,class U>
class aggregateSet
{
        friend setIterator<T,U,aggregateSet>;
public:
        void add(T a)
        {
                m_data.insert(a);
        }

        setIterator<T,U,aggregateSet>* create_iterator()
        {
                return new setIterator<T,U,aggregateSet>(this);
        }

        void Print()
        {
                copy(m_data.begin(),m_data.end(),ostream_iterator<T>(cout,"\r\n"));
        }

protected:
private:
```

```
        std::set<T,U> m_data;
};

#endif
/************************************************************************/
/* Iterator Test.cpp                                                    */
/************************************************************************/
#include <iostream>
#include <vector>
using namespace std;

class Money
{
public:
        Money(int a=0):m_data(a){}

        void SetMoney(int a){
                m_data = a;
        }

        int GetMoney()
        {
                return m_data;
        }

protected:
private:
        int m_data;
};

class Name
{
public:
        Name(string name):m_name(name){}

        const string& GetName() const{
                return m_name;
        }

        friend ostream operator<<(ostream cout,Name name)
        {
                cout<<name.GetName();
                return cout;
        }

private:
        string m_name;
};

struct NameLess
{
        bool operator() ( const Name& lhs, const Name& rhs) const
        {
                return lhs.GetName() < rhs.GetName();
        }
};

void main( void ) {
                //sample 1
        cout<<"_________________Iterator with
int_____________________________________"<<endl;
        aggregate<int> agg;

        for(int i=0;i<10;i++)
        {
                agg.add(i);
        }


        Iterator< int,aggregate<int> > *it = agg.create_iterator();
        for(it->first();!it->isDone();it->next())
```

```
        {
                cout<<*it->current()<<endl;
        }


        //sample 2
        aggregate<Money> agg2;
        Money a(100),b(1000),c(10000);
        agg2.add(a);
        agg2.add(b);
        agg2.add(c);

        cout<<"________________Iterator with Class
Money_____________________________"<<endl;
        Iterator< Money,aggregate<Money> > *it2 = agg2.create_iterator();
        it2->first();
        while (!it2->isDone()) {
                cout<<((Money*)(it2->current()))->GetMoney()<<endl;
                it2->next();
        }

        //sample 3
        cout<<"________________Set Iterator with Class
Name_____________________________"<<endl;

        aggregateSet<Name,NameLess> aset;
        aset.add(Name("Qmt"));
        aset.add(Name("Bmt"));
        aset.add(Name("Cmt"));
        aset.add(Name("Amt"));


        setIterator<Name,NameLess,aggregateSet<Name,NameLess>  > * it3 =
aset.create_iterator();
        for(it3->first();!it3->isDone();it3->next())
        {
                cout<<(*it3->current())<<endl;
        }
}
```

Mediator pattern

From Wikipedia, the free encyclopedia

Jump to: navigation, search

The **mediator pattern** is a software design pattern that provides a unified interface to a set of interfaces in a subsystem. It's one of the 23 design patterns described by the Gang of Four. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.

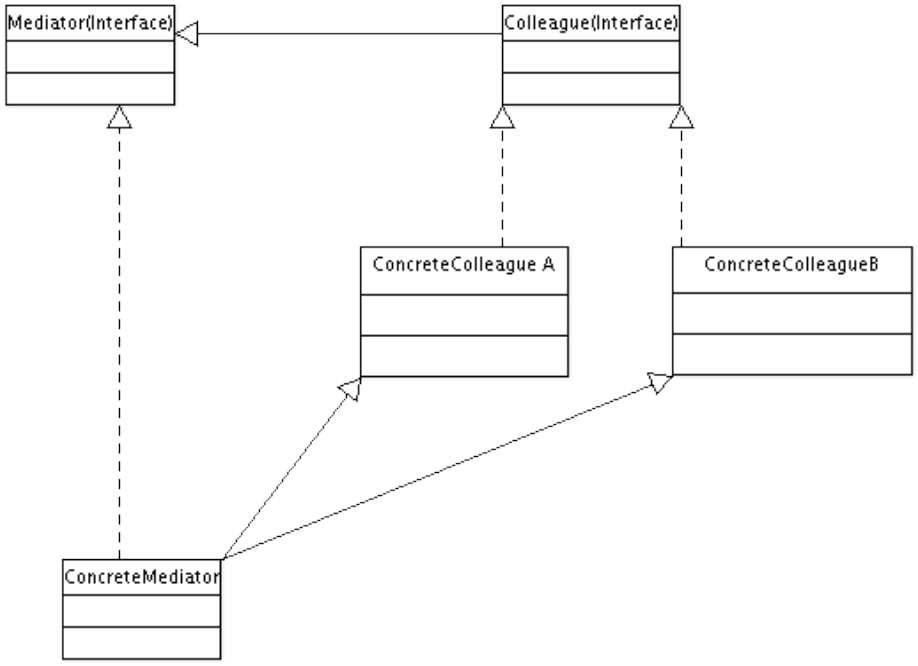
Usually a program is made up of a (sometimes large) number of classes. So the logic and computation is distributed among these classes. However, as more classes are developed in a program, especially during maintenance and/or refactoring, the problem of communication between these classes may become more complex. This makes the program harder to read and maintain. Furthermore, it can become difficult to change the program, since any change may affect code in several other classes.

With the **mediator pattern** communication between objects is encapsulated with a **mediator** object. Objects no longer communicate directly with each other, but instead

communicate through the mediator. This reduces the dependencies between communicating objects, thereby lowering the coupling.

Structure

The class diagram of this design pattern is as shown below:



Participants

Mediator - defines the interface for communication between *Colleague* objects

ConcreteMediator - implements the Mediator interface and coordinates communication between *Colleague* objects. It is aware of all the *Colleagues* and their purpose with regards to inter communication.

ConcreteColleague - communicates with other *Colleagues* through its *Mediator*

Memento pattern

From Wikipedia, the free encyclopedia

Jump to: navigation, search

The **memento pattern** is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

The memento pattern is used by two objects: the *originator* and a *caretaker*. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker can not, or should not, change). When using this pattern, care should be taken if the originator may change other objects or resources - the memento pattern operates on a single object.

Classic examples of the memento pattern include the seed of a pseudorandom number generator and the state in a finite state machine.

Actionscript 3

The folowing Actionscript 3 program illustrates the Memento Pattern.

```
package
{
	import flash.display.Sprite;

	/**
	* Memento
	*/
	public class As3Memento extends Sprite
	{
		function As3Memento()
		{
			var ct:CareTaker = new CareTaker();
			var originator:Originator = new Originator();

				originator.A = 'letter A';
				originator.B = 'letter B';
				originator.C = 'letter C';

				ct.addMementoFrom(originator);

				originator.A = 'anything...';
				originator.B = 'blah blah...';
				originator.C = 'etc...';
				ct.addMementoFrom(originator);

				originator.restoreFromMemento(ct.getMemento(0));

				trace(originator.A, originator.B, originator.C);

				originator.restoreFromMemento(ct.getMemento(1));

				trace(originator.A, originator.B, originator.C);

		}
	}

}

interface IMemento
{
```

```
        function getMemento():Object;
        function restoreFromMemento(obj:Object):void;
}

class Originator implements IMemento
{
        public var A:String;
        public var B:String;
        public var C:String;

        function Originator()
        { }

        /* INTERFACE IMemento */

        public function getMemento():Object
        {
                return { _A:A, _B:B, _C:C };
        }

        public function restoreFromMemento(obj:Object):void
        {
                A = obj._A;
                B = obj._B;
                C = obj._C;
        }

}

class CareTaker
{
        private var mementos:Array;
        function CareTaker()
        {
                mementos = new Array();
        }

        public function addMementoFrom( object:IMemento ):void
        {
                mementos.push( object.getMemento() );
        }

        public function getMemento(index:uint):Object
        {
                return mementos[index];
        }
}
```

Java

The following Java program illustrates the "undo" usage of the Memento Pattern.

```
import java.util.*;

class Originator {
    private String state;
    /* lots of memory consumptive private data that is not necessary to define the
     * state and should thus not be saved. Hence the small memento object. */

    public void set(String state) {
        System.out.println("Originator: Setting state to "+state);
        this.state = state;
    }

    public Object saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }
    public void restoreFromMemento(Object m) {
```

```
        if (m instanceof Memento) {
            Memento memento = (Memento)m;
            state = memento.getSavedState();
            System.out.println("Originator: State after restoring from Memento: "+state);
        }
    }

    private static class Memento {
        private String state;

        public Memento(String stateToSave) { state = stateToSave; }
        public String getSavedState() { return state; }
    }
}

class Caretaker {
    private List<Object> savedStates = new ArrayList<Object>();

    public void addMemento(Object m) { savedStates.add(m); }
    public Object getMemento(int index) { return savedStates.get(index); }
}

class MementoExample {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State3");
        caretaker.addMemento( originator.saveToMemento() );
        originator.set("State4");

        originator.restoreFromMemento( caretaker.getMemento(1) );
    }
}
```

The output is:

```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```

Ruby

The following Ruby program illustrates the same pattern.

```
#!/usr/bin/env ruby -KU

require 'rubygems'
require 'spec'

class Originator
  class Memento
    def initialize(state)
      # dup required so that munging of the Originator's original state doesn't mess up
      # this Memento for first or subsequent restore
      @state = state.dup
    end

    def state
      # dup required so that munging of the Originator's restored state doesn't mess up
```

```
      # this Memento for a second restore
      @state.dup
    end
  end

  attr_accessor :state

  # pretend there's lots of additional memory-heavy data, which can be reconstructed
  # from state
  def save_to_memento
    Memento.new(@state)
  end

  def restore_from_memento(m)
    @state = m.state
  end

end

class Caretaker < Array; end

describe Originator do
  before(:all) do
    @caretaker = Caretaker.new
    @originator = Originator.new

    @originator.state = "State1"
  end

  if "should have original state" do
    @originator.state.should == 'State1'
  end

  if "should update state" do
    @originator.state = "State2"
    @originator.state.should == 'State2'
  end

  if "should save memento" do
    @caretaker << @originator.save_to_memento
    @caretaker.size.should == 1
  end

  if "should update state after save to memento" do
    @originator.state = "State3"
    @originator.state.should == 'State3'
  end

  if "should save to memento again" do
    @caretaker << @originator.save_to_memento
    @caretaker.size.should == 2
  end

  if "should update state after save to memento again" do
    @originator.state = "State4";
    @originator.state.should == 'State4'
  end

  if "should restore to original save point" do
    @originator.restore_from_memento @caretaker[0]
    @originator.state.should == 'State2'
  end

  if "should restore to second save point" do
    @originator.restore_from_memento @caretaker[1]
    @originator.state.should == 'State3'
  end

  if "should restore after pathological munging of restored state" do
    @originator.state[-1] = '5'
    @originator.state.should == 'State5'
```

```
    @originator.restore_from_memento @caretaker[1]
    @originator.state.should == 'State3'
  end

  if "should restore after pathological munging of original state" do
    @originator.state = "State6"
    @originator.state.should == 'State6'
    @caretaker << @originator.save_to_memento
    @originator.state[-1] = '7'
    @originator.state.should == 'State7'
    @originator.restore_from_memento @caretaker[2]
    @originator.state.should == 'State6'
  end
end
```

Observer pattern

The **observer pattern** (sometimes known as publish/subscribe) is a design pattern used in computer programming to observe the state of an object in a program. It is related to the principle of implicit invocation.

This pattern is mainly used to implement a distributed event handling system. In some programming languages, the issues addressed by this pattern are handled in the native event handling syntax. This is a very interesting feature in terms of real-time deployment of applications.

The primary objective of this pattern is to provide a way to handle runtime one-to-many relationships between objects in object oriented languages. The event from the observable object's point of view is called **notification** and the event from the observers' point of view is called **update**.

A minimal way how to extend this principle is the usage of two one-to-many relationships, where the second one is bound to observers of the first one. This principle is called Model-view-controller.

More theory on event driven systems is described in article Petri net.

Overview

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. To achieve this one object (subject) should know about its dependents. Subject maintains list of its dependents. Each dependent who wants to get notification on subject state change, should register with subject.

Participant classes

The participants of the pattern are detailed below. Member functions are listed with bullets.

Subject

This abstract class provides an interface for attaching and detaching observers. Subject class also holds a private list of observers. Contains these functions (methods):

> *Attach* - Adds a new observer to the list of observers observing the subject.
> *Detach* - Removes an existing observer from the list of observers observing the subject
> *Notify* - Notifies each observer by calling the update() function in the observer, when a change occurs.

ConcreteSubject

This class provides the state of interest to observers. It also sends a notification to all observers, by calling the Notify function in its superclass or base class (i.e, in the Subject class). Contains this function:

> *GetState* - Returns the state of the subject.

Observer

This class defines an **updating interface for all observers**, to receive update notification from the subject. The Observer class is used as an abstract class to implement concrete observers. Contains this function:

> *Update* - An abstract function, to be **overridden** by concrete **observers**.

ConcreteObserver

This class maintains a reference with the ConcreteSubject, to receive the state of the subject when a notification is received. Contains this function:

> *Update* - This is the overridden function in the concrete class. When this function is called by the subject, the ConcreteObserver calls the GetState function of the subject to update the information it has about the subject's state.

When the event is raised each observer receives a callback. This may be either a virtual function of the observer class (called 'update()' on the diagram) or a function pointer (more generally a function object or "functor") passed as an argument to the listener registration method. The update function may also be passed some parameters (generally information about the event that is occurring) which can be used by the observer. When an event occurs within the subject, the subject then calls the update method on all observers attached to that subject.

Each concrete observer implements the update function and as a consequence defines its own behavior when the notification occurs.

Typical usages

> When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects allows the programmer to vary and reuse them independently.
> When a change to one object requires changing others, and it's not known in advance how many objects need to be changed.
> When an object should be able to notify other objects without making assumptions about who these objects are.

The observer pattern is also very often associated with the model-view-controller (MVC) paradigm. In MVC, the observer pattern is used to create a loose coupling between the model and the view. Typically, a modification in the model triggers the notification of model observers which are actually the views.

An example is Java Swing, in which the model is expected to issue change notifications to the views via the *PropertyChangeNotification* infrastructure. Model classes are Java beans that behave as the subject, described above. View classes are associated with some visible item on the GUI and behave as the observers, described above. As the application runs, changes are made to the model. The user sees these changes because the views are updated accordingly.

C#

```
using System;
using System.Collections;

namespace Wikipedia.Patterns.Strategy
{
        // IObserver --> interface for the observer
        public interface IObserver
        {
                // called by the subject to update the observer of any change
                // The method parameters can be modified to fit certain criteria
                void Update(string message);
        }

        public class Subject
        {
                // use array list implementation for collection of observers
                private ArrayList observers;

                // constructor
                public Subject()
                {
                        observers = new ArrayList();
                }

                public void Register(IObserver observer)
                {
                        // if list does not contain observer, add
                        if (!observers.Contains(observer))
                        {
                                observers.Add(observer);
                        }
                }

                public void Unregister(IObserver observer)
                {
```

```
                        // if observer is in the list, remove
                        if (observers.Contains(observer))
                        {
                                observers.Remove(observer);
                        }
                }

                public void Notify(string message)
                {
                        // call update method for every observer
                        foreach (IObserver observer in observers)
                        {
                                observer.Update(message);
                        }
                }
        }

        // Observer1 --> Implements the IObserver
        public class Observer1 : IObserver
        {
                public void Update(string message)
                {
                        Console.WriteLine("Observer1:" + message);
                }
        }

        // Observer2 --> Implements the IObserver
        public class Observer2 : IObserver
        {
                public void Update(string message)
                {
                        Console.WriteLine("Observer2:" + message);
                }
        }

        // Test class
        public class ObserverTester
        {
                [STAThread]
                public static void Main()
                {
                        Subject mySubject = new Subject();
                        IObserver myObserver1 = new Observer1();
                        IObserver myObserver2 = new Observer2();

                        // register observers
                        mySubject.Register(myObserver1);
                        mySubject.Register(myObserver2);

                        mySubject.Notify("message 1");
                        mySubject.Notify("message 2");
                }
        }
}
```

Python

The observer pattern in Python:

```
class AbstractSubject:
    def register(self, listener):
        raise NotImplementedError, "Must subclass me"

    def unregister(self, listener):
        raise NotImplementedError, "Must subclass me"

    def notify_listeners(self, event):
        raise NotImplementedError, "Must subclass me"
```

```
class Listener:
    def __init__(self, name, subject):
        self.name = name
        subject.register(self)

    def notify(self, event):
        print self.name, "received event", event

class Subject(AbstractSubject):
    def __init__(self):
        self.listeners = []
        self.data = None

    def getUserAction(self):
        self.data = raw_input('Enter something to do:')
        return self.data

    # Implement abstract Class AbstractSubject

    def register(self, listener):
        self.listeners.append(listener)

    def unregister(self, listener):
        self.listeners.remove(listener)

    def notify_listeners(self, event):
        for listener in self.listeners:
            listener.notify(event)


if __name__=="__main__":
    # make a subject object to spy on
    subject = Subject()

    # register two listeners to monitor it.
    listenerA = Listener("<listener A>", subject)
    listenerB = Listener("<listener B>", subject)

    # simulated event
    subject.notify_listeners ("<event 1>")
    # outputs:
    #     <listener A> received event <event 1>
    #     <listener B> received event <event 1>

    action = subject.getUserAction()
    subject.notify_listeners(action)
    #Enter something to do:hello
    # outputs:
    #     <listener A> received event hello
    #     <listener B> received event hello
```

The observer pattern can be implemented more succinctly in Python using function decorators.

Java

Here is an example that takes keyboard input and treats each input line as an event. The example is built upon the library classes `java.util.Observer` and `java.util.Observable`. When a string is supplied from System.in, the method `notifyObserver` is then called, in order to notify all observers of the event's occurrence, in the form of an invocation of their 'update' methods - in our example, `ResponseHandler.update(...)`.

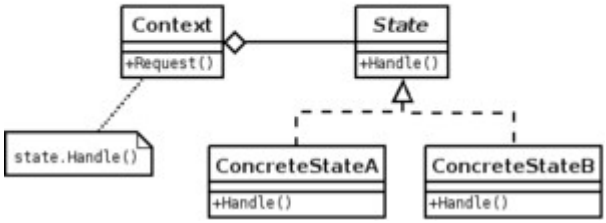
The file myapp.java contains a main() method that might be used in order to run the code.

```
/* File Name : EventSource.java */

package obs;
import java.util.Observable;          //Observable is here
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EventSource extends Observable implements Runnable
{
    public void run()
    {
        try
        {
            final InputStreamReader isr = new InputStreamReader( System.in );
            final BufferedReader br = new BufferedReader( isr );
            while( true )
            {
                final String response = br.readLine();
                setChanged();
                notifyObservers( response );
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
/* File Name: ResponseHandler.java */

package obs;

import java.util.Observable;
import java.util.Observer;  /* this is Event Handler */

public class ResponseHandler implements Observer
{
    private String resp;
    public void update (Observable obj, Object arg)
    {
        if (arg instanceof String)
        {
            resp = (String) arg;
            System.out.println("\nReceived Response: "+ resp );
        }
    }
}
/* Filename : myapp.java */
/* This is main program */

package obs;

public class MyApp
{
    public static void main(String args[])
    {
        System.out.println("Enter Text >");

        // create an event source - reads from stdin
        final EventSource evSrc = new EventSource();

        // create an observer
        final ResponseHandler respHandler = new ResponseHandler();

        // subscribe the observer to the event source
        evSrc.addObserver( respHandler );
```

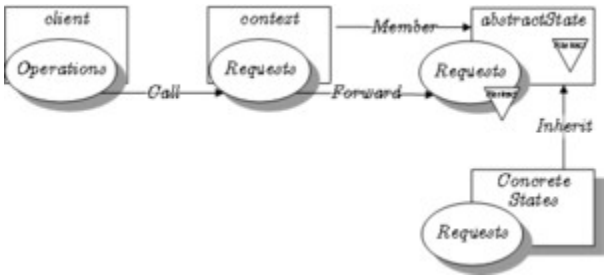
```
        // starts the event thread
        Thread thread = new Thread(evSrc);
        thread.start();
    }
}
```

Also refer to an article in JavaWorld http://www.javaworld.com/javaworld/javaqa/2001-05/04-qa-0525-observer.html

State pattern



State in UML



State in LePUS3 (legend)

The **state pattern** is a behavioral software design pattern, also known as the **objects for states pattern**. This pattern is used in computer programming to represent the state of an object. This is a clean way for an object to partially change its type at runtime[1].

Take for example, a drawing program, in which there could be an abstract interface representing a tool, then concrete instances of that class could each represent a kind of tool. When the user selects a different tool, the appropriate tool would be instantiated.

For example, an interface to a drawing tool could be

```
class AbstractTool {
public:
  virtual void MoveTo(const Point& inP) = 0;
  virtual void MouseDown(const Point& inP) = 0;
  virtual void MouseUp(const Point& inP) = 0;
};
```

Then a simple pen tool could be

```
class PenTool : public AbstractTool {
public:
  PenTool() : mMouseIsDown(false) {}
  virtual void MoveTo(const Point& inP) {
    if(mMouseIsDown) {
      DrawLine(mLastP, inP);
    }
    mLastP = inP;
  }
  virtual void MouseDown(const Point& inP) {
    mMouseIsDown = true;
    mLastP = inP;
  }
  virtual void MouseUp(const Point& inP) {
    mMouseIsDown = false;
  }
private:
  bool mMouseIsDown;
  Point mLastP;
};

class SelectionTool : public AbstractTool {
public:
  SelectionTool() : mMouseIsDown(false) {}
  virtual void MoveTo(const Point& inP) {
    if(mMouseIsDown) {
      mSelection.Set(mLastP, inP);
    }
  }
  virtual void MouseDown(const Point& inP) {
    mMouseIsDown = true;
    mLastP = inP;
    mSelection.Set(mLastP, inP);
  }
  virtual void MouseUp(const Point& inP) {
    mMouseIsDown = false;
  }
private:
  bool mMouseIsDown;
  Point mLastP;
  Rectangle mSelection;
};
```

A client using the state pattern above could look like this:

```
class DrawingController {
 public:
  DrawingController() { selectPenTool(); } // Start with some tool.
  void MoveTo(const Point& inP) {currentTool->MoveTo(inP)}
  void MouseDown(const Point& inP) {currentTool->MouseDown(inP)}
  void MouseUp(const Point& inP) {currentTool->MouseUp(inP)}

  selectPenTool() {
    currentTool.reset(new PenTool);
  }

  selectSelectionTool() {
    currentTool.reset(new SelectionTool);
  }

 private:
  std::auto_ptr<AbstractTool> currentTool;
};
```

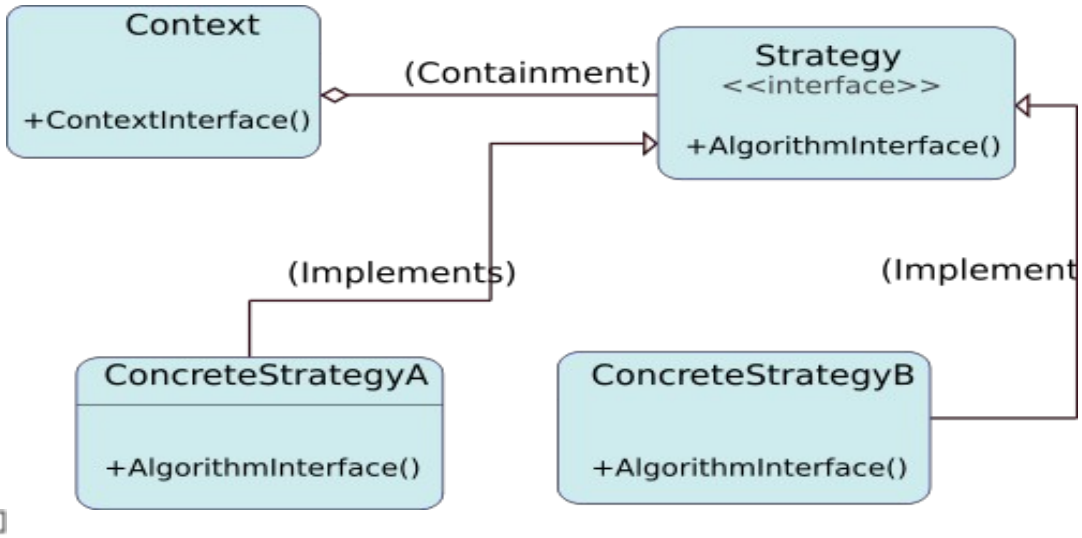
The state of the drawing tool is thus represented entirely by an instance of AbstractTool. This makes it easy to add more tools and to keep their behavior localized to that subclass of `AbstractTool`.

As opposed to using switch

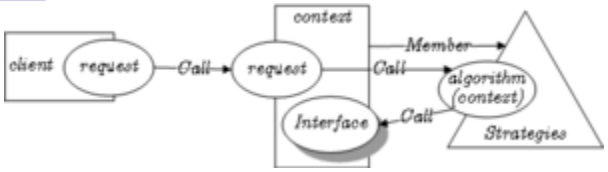
The state pattern can be used to replace `switch()` statements and `if {}` statements which can be difficult to maintain and are less type-safe. For example, the following is similar to the above but adding a new tool type to this version would be much more difficult.

```
class Tool {
public:
  Tool() : mMouseIsDown(false) {}
  virtual void MoveTo(const Point& inP);
  virtual void MouseDown(const Point& inP);
  virtual void MouseUp(const Point& inP);
private:
  enum Mode { Pen, Selection };
  Mode mMode;
  Point mLastP;
  bool mMouseIsDown;
  Rectangle mSelection;
};

void Tool::MoveTo(const Point& inP) {
  switch(mMode) {
  case Pen:
    if(mMouseIsDown) {
      DrawLine(mLastP, inP);
    }
    mLastP = inP;
    break;
  case Selection:
    if(mMouseIsDown) {
      mSelection.Set(mLastP, inP);
    }
    break;
  default:
    throw std::exception();
  }
}

void Tool::MouseDown(const Point& inP) {
  switch(mMode) {
  case Pen:
    mMouseIsDown = true;
    mLastP = inP;
    break;
  case Selection:
    mMouseIsDown = true;
    mLastP = inP;
    mSelection.Set(mLastP, inP);
    break;
  default:
    throw std::exception();
  }
}

void Tool::MouseUp(const Point& inP) {
  mMouseIsDown = false;
}
```

Strategy pattern



Strategy Pattern in UML



Strategy pattern in LePUS3 (legend)

In computer programming, the **strategy pattern** (also known as the **policy pattern**) is a particular software design pattern, whereby algorithms can be selected at runtime.

In some programming languages, such as those without polymorphism, the issues addressed by this pattern are handled through forms of reflection, such as the native function pointer or function delegate syntax.

This pattern is invisible in languages with first-class functions. See the Python code for an example.

The strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application. The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them.

Code Examples

Java

```
package wikipedia.patterns.strategy;
```

```
//StrategyExample test application
public class StrategyExample {

  public static void main(String[] args) {
    Context context;

    // Three contexts following different strategies
    context = new Context(new ConcreteStrategyA());
    context.execute();

    context = new Context(new ConcreteStrategyB());
    context.execute();

    context = new Context(new ConcreteStrategyC());
    context.execute();
  }
}
  // The classes that implement a concrete strategy should implement this
  // The context class uses this to call the concrete strategy
  interface IStrategy {
    void execute();
  }

  // Implements the algorithm using the strategy interface
  class ConcreteStrategyA implements IStrategy {
    public void execute() {
      System.out.println( "Called ConcreteStrategyA.execute()" );
    }
  }

  class ConcreteStrategyB implements IStrategy  {
    public void execute() {
      System.out.println( "Called ConcreteStrategyB.execute()" );
    }
  }

  class ConcreteStrategyC implements IStrategy {
    public void execute() {
      System.out.println( "Called ConcreteStrategyC.execute()" );
    }
  }

  // Configured with a ConcreteStrategy object and maintains a reference to a Strategy
object
  class Context {
    IStrategy strategy;

    // Constructor
    public Context(IStrategy strategy) {
      this.strategy = strategy;
    }

    public void execute() {
      this.strategy.execute();
    }
  }
```

Python

Python has first-class functions, so there's no need to implement this pattern explicitly. However one loses information because the interface of the strategy is not made explicit. Here's an example you might encounter in GUI programming, using a callback function:

```
class Button:
    """A very basic button widget."""
    def __init__(self, submit_func, label):
        self.on_submit = submit_func   # Set the strategy function directly
```

```
        self.label = label

# Create two instances with different strategies
button1 = Button(sum, "Add 'em")
button2 = Button(lambda nums: " ".join(map(str, nums)), "Join 'em")

# Test each button
numbers = range(1, 10)   # A list of numbers 1 through 9
print button1.on_submit(numbers)   # displays "45"
print button2.on_submit(numbers)   # displays "1 2 3 4 5 6 7 8 9"
```

C#

```
using System;

namespace Wikipedia.Patterns.Strategy
{
  // MainApp test application
  class MainApp
  {
    static void Main()
    {
      Context context;

      // Three contexts following different strategies
      context = new Context(new ConcreteStrategyA());
      context.Execute();

      context = new Context(new ConcreteStrategyB());
      context.Execute();

      context = new Context(new ConcreteStrategyC());
      context.Execute();

    }
  }

  // The classes that implement a concrete strategy should implement this
  // The context class uses this to call the concrete strategy
  interface IStrategy
  {
    void Execute();
  }

  // Implements the algorithm using the strategy interface
  class ConcreteStrategyA : IStrategy
  {
    public void Execute()
    {
      Console.WriteLine( "Called ConcreteStrategyA.Execute()" );
    }
  }

  class ConcreteStrategyB : IStrategy
  {
    public void Execute()
    {
      Console.WriteLine( "Called ConcreteStrategyB.Execute()" );
    }
  }

  class ConcreteStrategyC : IStrategy
  {
    public void Execute()
    {
      Console.WriteLine( "Called ConcreteStrategyC.Execute()" );
    }
  }
```

```
  // Configured with a ConcreteStrategy object and maintains a reference to a Strategy
object
  class Context
  {
    IStrategy strategy;

    // Constructor
    public Context(IStrategy strategy)
    {
      this.strategy = strategy;
    }

    public void Execute()
    {
      strategy.Execute();
    }
  }
}
```

ActionScript 3

```
//invoked from application.initialize
private function init() : void
{
    var context:Context;

    context = new Context( new ConcreteStrategyA() );
    context.execute();

    context = new Context( new ConcreteStrategyB() );
    context.execute();

    context = new Context( new ConcreteStrategyC() );
    context.execute();
}

package org.wikipedia.patterns.strategy
{
    public interface IStrategy
    {
        function execute() : void ;
    }
}

package org.wikipedia.patterns.strategy
{
    public final class ConcreteStrategyA implements IStrategy
    {
        public function execute():void
        {
             trace( "ConcreteStrategyA.execute(); invoked" );
        }
    }
}

package org.wikipedia.patterns.strategy
{
    public final class ConcreteStrategyB implements IStrategy
    {
        public function execute():void
        {
             trace( "ConcreteStrategyB.execute(); invoked" );
        }
    }
}

package org.wikipedia.patterns.strategy
{
    public final class ConcreteStrategyC implements IStrategy
    {
        public function execute():void
```

```
        {
            trace( "ConcreteStrategyC.execute(); invoked" );
        }
    }
}

package org.wikipedia.patterns.strategy
{
   public class Context
   {
        private var strategy:IStrategy;

        public function Context(strategy:IStrategy)
        {
            this.strategy = strategy;
        }

        public function execute() : void
        {
            strategy.execute();
        }
    }
}
```

PHP

```
<?php
class StrategyExample {
    public function __construct() {
        $context = new Context(new ConcreteStrategyA());
        $context->execute();

        $context = new Context(new ConcreteStrategyB());
        $context->execute();

        $context = new Context(new ConcreteStrategyC());
        $context->execute();
    }
}

interface IStrategy {
    public function execute();
}

class ConcreteStrategyA implements IStrategy {
    public function execute() {
        echo "Called ConcreteStrategyA execute method\n";
    }
}

class ConcreteStrategyB implements IStrategy {
    public function execute() {
        echo "Called ConcreteStrategyB execute method\n";
    }
}

class ConcreteStrategyC implements IStrategy {
    public function execute() {
        echo "Called ConcreteStrategyC execute method\n";
    }
}

class Context {
    var $strategy;

    public function __construct(IStrategy $strategy) {
        $this->strategy = $strategy;
    }

    public function execute() {
        $this->strategy->execute();
```

```
    }
}

new StrategyExample;
?>
```

Strategy versus Bridge

The UML class diagram for the Strategy pattern is the same as the diagram for the Bridge pattern. However, these two design patterns aren't the same in their *intent*. While the Strategy pattern is meant for *behavior*, the Bridge pattern is meant for *structure*.

The coupling between the context and the strategies is tighter than the coupling between the abstraction and the implementation in the Bridge pattern.

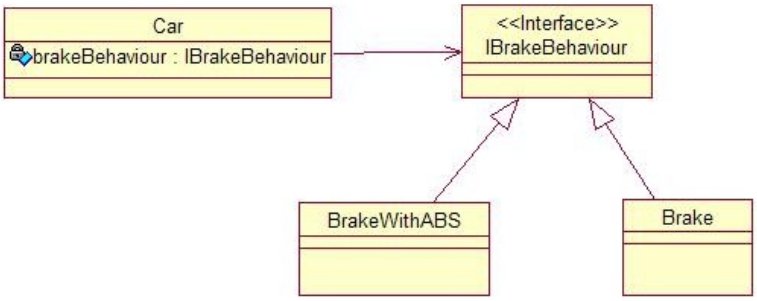
Strategy Pattern and Open Closed Principle

According to Strategy pattern, the behaviors of a class should not be inherited, instead they should be encapsulated using interfaces. As an example, consider a car class. Two possible behaviors of car are brake and accelerate.

Since accelerate and brake behaviors change frequently between models, a common approach is to implement these behaviors in subclasses. This approach has significant drawbacks: accelerate and brake behaviors must be declared in each new Car model. This may not be a concern when there are only a small number of models, but the work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.

The strategy pattern uses composition instead of inheritance. In the strategy pattern behaviors are defined as separate interfaces and specific classes that implement these interfaces. Specific classes encapsulate these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can also be changed at run-time as well as at design-time. For instance, a car object's brake behavior can be changed from BrakeWithABS() to Brake() by changing the brakeBehavior member to:

brakeBehavior = new Brake();

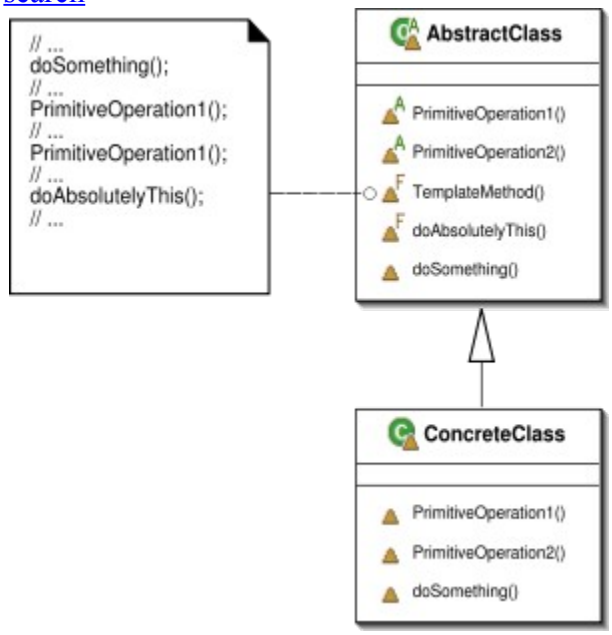


This gives greater flexibility in design and is in harmony with the Open/closed principle (OCP) that states classes should be open for extension but closed for modification.

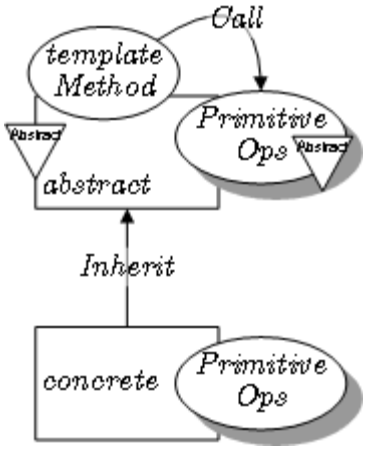
Template method pattern

From Wikipedia, the free encyclopedia

Jump to: navigation, search



Template method: UML class diagram.



Template Method in LePUS3 (legend)

In software engineering, the **template method pattern** is a design pattern. It is a so-called behavioral pattern, and is unrelated to C++ templates.

Introduction

In a template pattern, the model (data such as XML, or a set of APIs) has no inherent knowledge of how it would be utilized. The actual algorithm is delegated to the views, i.e. templates. Different templates could be applied to the same set of data or APIs and produce different results. Thus, it is a subset of model-view-controller patterns without the controller. The pattern does not have to be limited to the object-oriented programming. For example, different XSLT templates could render the same XML data and produce different outputs. C++ templates also make it possible to code the algorithms (i.e. views) without having to use abstract classes or interfaces.

In object-oriented programming, first a class is created that provides the basic steps of an algorithm design. These steps are implemented using abstract methods. Later on, subclasses change the abstract methods to implement real actions. Thus the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.

The template method thus manages the larger picture of task semantics, and more refined implementation details of selection and sequence of methods. This larger picture calls abstract and non-abstract methods for the task at hand. The non-abstract methods are completely controlled by the Template method. The expressive power and degrees of freedom occur in abstract methods that may be implemented in subclasses. Some or all of the abstract methods can be specialized in the subclass, the abstract method is the smallest unit of granularity, allowing the writer of the subclass to provide particular behavior with minimal modifications to the larger semantics. In contrast the template method need not be changed and is not an abstract operation and thus may guarantee required steps before and after the abstract operations. Thus the template method is invoked and as a consequence the subordinate non-abstract methods and abstract methods are called in the correct sequence.

The template method occurs frequently, at least in its simplest case, where a method calls only one abstract method, with object oriented languages. If a software writer uses a polymorphic method at all, this design pattern may be a rather natural consequence. This is because a method calling an abstract or polymorphic function is simply the reason for being of the abstract or polymorphic method. The template method may be used to add immediate present value to the software or with a vision to enhancements in the future.

The template method is strongly related to the NVI (Non-Virtual Interface) pattern. The NVI pattern recognizes the benefits of a non-abstract method invoking the subordinate abstract methods. This level of indirection allows for pre and post operations relative to the abstract operations both immediately and with future unforeseen changes. The NVI pattern can be deployed with very little software production and runtime cost. Many commercial frameworks employ the NVI pattern.

Usage

The template method is used to:

- let subclasses implement behaviour that can vary
- avoid duplication in the code: you look for the general code in the algorithm, and implement the variants in the subclasses
- control at what point(s) subclassing is allowed.

The control structure (inversion of control) that is the result of the application of a template pattern is often referred to as the Hollywood Principle: "Don't call us, we'll call you." Using this principle, the template method in a parent class controls the overall process by calling subclass methods as required. This is shown in the following example:

Example (in Java)

```
/**
 * An abstract class that is common to several games in
 * which players play against the others, but only one is
 * playing at a given time.
 */

abstract class Game {

    private int playersCount;

    abstract void initializeGame();

    abstract void makePlay(int player);

    abstract boolean endOfGame();

    abstract void printWinner();

    /* A template method : */
    final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()){
```

```
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
 }

//Now we can extend this class in order to implement actual games:

 class Monopoly extends Game {

     /* Implementation of necessary concrete methods */

     void initializeGame() {
         // ...
     }

     void makePlay(int player) {
         // ...
     }

     boolean endOfGame() {
         // ...
     }

     void printWinner() {
         // ...
     }

     /* Specific declarations for the Monopoly game. */

     // ...

 }

 class Chess extends Game {

     /* Implementation of necessary concrete methods */

     void initializeGame() {
         // ...
     }

     void makePlay(int player) {
         // ...
     }

     boolean endOfGame() {
         // ...
     }

     void printWinner() {
         // ...
     }

     /* Specific declarations for the chess game. */

     // ...

 }
```

Example (in C++)

In C++ it is not necessary to use inheritance and abstract (pure virtual) methods. For example, the Standard Library (STL) has been conceived using another paradigm: the templatization. Here is an example using the "traits" feature on which the STL relies.

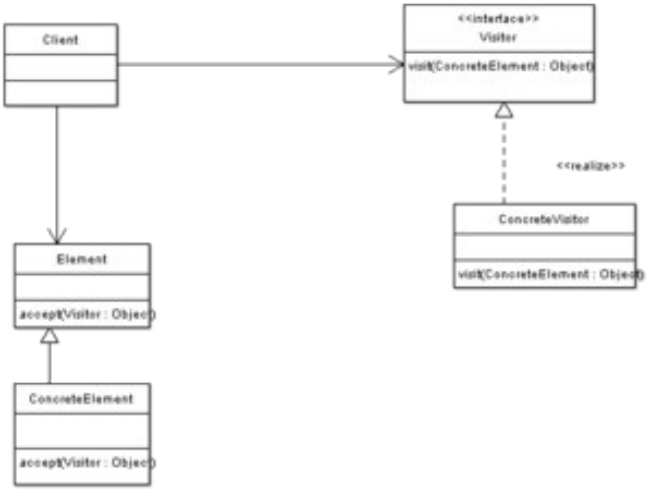
```
   /**
  * The use of traits class to implement the Template Method Pattern
  * Note: this method has been heavily used in the Standard Library, the example below is
just a bit simpler
  */

  "example.h"

  template <class T>
  class example_traits
  {
  public:
    /**
    * Typedefs
    */
    typedef T value;
    typedef const T const_value;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;

    /**
    * Methods
    */
    static pointer Clone(const_pointer iClonee)
    {
      return new T(*iClonee);
    } // Clone
  }; // class example_traits

  template <class T, class U = example_traits<T> >
  class example
  {
  public:
    /**
    * Typedefs
    */
    typedef typename U::value value;
    typedef typename U::const_value const_value;
    typedef typename U::pointer pointer;
    typedef typename U::const_pointer const_pointer;
    typedef typename U::reference reference;
    typedef typename U::const_reference const_reference

    /**
    * Default constructors and such are not shown
    */
    // We just add a constructor that takes a pointer to the object to initialize its
attribute
    example(pointer object); // definition not shown

    /**
    * Methods
    */
    example clone()
    {

      // Some code to log that we clone the object

      pointer aClone = U::Clone(object_); // actual cloning

      // Some code to clean up the object or change its state if we want

      return example(aClone);
    } // clone

  private:
    /**
    * object
    */
```

```
    pointer object_;
  }; // example

  "myClass.h"

  class MyClass
  {
  public:
    MyClass* duplicateMe() const; // method to obtain a clone
    /** stuff **/
  }; // MyClass

  /**
  * Two different methods
  */

  // specialization of the traits class
  template <>
  class example_traits<MyClass>
  {
    /**
    * Typedefs
    */
    typedef MyClass value;
    typedef const MyClass const_value;
    typedef MyClass* pointer;
    typedef const MyClass* const_pointer;
    typedef MyClass& reference;
    typedef const MyClass& const_reference;

    /**
    * Methods
    */
    static pointer Clone(const_pointer iClonee)
    {
      return iClonee->duplicateMe(); // This line is changed to use the clone method of
MyClass
    } // Clone
  }; // example<MyClass>

  // creation of a brand new class
  class MyClassExampleTraits
  {
    /**
    * Typedefs
    */
    typedef MyClass value;
    typedef const MyClass const_value;
    typedef MyClass* pointer;
    typedef const MyClass* const_pointer;
    typedef MyClass& reference;
    typedef const MyClass& const_reference;

    /**
    * Methods
    */
    static pointer Clone(const_pointer iClonee)
    {
      return iClonee->duplicateMe(); // This line is changed to use the clone method of
MyClass
    } // Clone
  }; // MyClassExampleTraits


  "main.cpp"

  int main()
  {
    // Using the first method
    example<MyClass> aFirstExample; // the default parameter is set to
example_traits<MyClass> which is our specialization
```

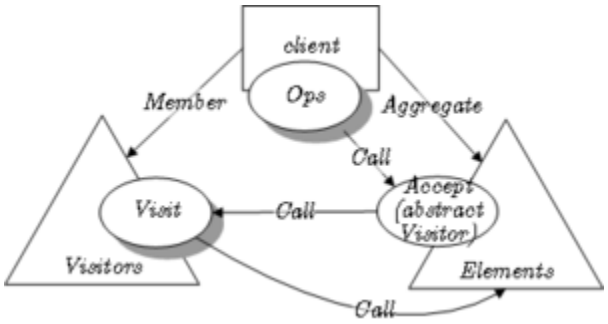
```
    example<MyClass> aFirstClone = aFirstExample->clone();

    // using the first method, but precising the second parameter
    example<MyClass, example_traits<MyClass> > aSecondExample;
    aFirstClone = aSecondExample->clone(); // Note that the type of aFirstClone is
compatible
                                           // in fact it is exactly the same type (we
have just explicitely precised the argument)

    // Using the second method
    example<MyClass, MyClassExampleTraits> aThirdExample; // the traits class is a user
defined one
    example<MyClass, MyClassExampleTraits> aSecondClone = aThirdExample->clone();

    return 0;
  } // main
```

Visitor pattern

Jump to: navigation, search



Visitor in UML



Visitor in LePUS3 (legend)

In object-oriented programming and software engineering, the **visitor** design pattern is a way of separating an algorithm from an object structure upon which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. Thus, using the visitor pattern helps conformance with the open/closed principle.

In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

While powerful, the visitor pattern does have limitations as compared with conventional virtual functions. It is not possible to create visitors for objects without adding a small callback method inside each class and the callback method in each of the classes is not inheritable to the level of the new subclass.

Elaborated

The idea is to use a structure of element classes, each of which has an `accept()` method that takes a `visitor` object as an argument. `Visitor` is an interface that has a `visit()` method for each element class. The `accept()` method of an element class calls back the `visit()` method for its class. Separate concrete `visitor` classes can then be written that perform some particular operations, by implementing these operations in their respective `visit()` methods.

One of these `visit()` methods of a concrete `visitor` can be thought of as methods not of a single class, but rather methods of a pair of classes: the concrete visitor and the particular element class. Thus the visitor pattern simulates double dispatch in a conventional single-dispatch object-oriented language such as Java, Smalltalk, and C++. For an explanation of how double dispatch differs from function overloading, see Double dispatch is more than function overloading in the double dispatch article. In the Java language, two techniques have been documented which use reflection to simplify the mechanics of double dispatch simulation in the visitor pattern: getting rid of accept() methods (the Walkabout variation), and getting rid of extra visit() methods.

The visitor pattern also specifies how iteration occurs over the object structure. In the simplest version, where each algorithm needs to iterate in the same way, the `accept()` method of a container element, in addition to calling back the `visit()` method of the `visitor`, also passes the `visitor` object to the `accept()` method of all its constituent child elements.

Because the Visitor object has one principal function (manifested in a plurality of specialized methods) and that function is called `visit()`, the Visitor can be readily identified as a potential function object or functor. Likewise, the `accept()` function can be identified as a function applicator, a mapper, which knows how to traverse a particular type of object and apply a function to its elements. Lisp's object system with its multiple dispatch does not replace the Visitor pattern, but merely provides a more concise implementation of it in which the pattern all but disappears.

The following example is an example in the Java programming language:

```
interface Visitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visitCar(Car car);
}
interface CarElement{
    public void accept(Visitor visitor);
}
class Wheel implements CarElement{
    private String name;
    Wheel(String name) {
        this.name = name;
    }
    String getName() {
        return this.name;
    }
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements CarElement{
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Body implements CarElement{
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Car  {
    CarElement[] elements;
    public CarElement [] getElements(){
        return elements.clone();
    }
    public Car() {
        this.elements = new CarElement[]
          { new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right"),
            new Body(), new Engine()};
    }
}

class PrintVisitor implements Visitor {

    public void visit(Wheel wheel) {
        System.out.println("Visiting "+ wheel.getName()
                           + " wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visitCar(Car car) {
        System.out.println("\nVisiting car");
        for(CarElement element : car.getElements()) {
            element.accept(this);
        }
        System.out.println("Visited car");
    }
}

class DoVisitor implements Visitor {
    public void visit(Wheel wheel) {
```

```
            System.out.println("Kicking my "+ wheel.getName());
        }
        public void visit(Engine engine) {
            System.out.println("Starting my engine");
        }
        public void visit(Body body) {
            System.out.println("Moving my body");
        }
        public void visitCar(Car car) {
            System.out.println("\nStarting my car");
            for(CarElement carElement : car.getElements()) {
                carElement.accept(this);
            }
            System.out.println("Started car");
        }

    }

    public class VisitorDemo {
        static public void main(String[] args){
            Car car = new Car();
            Visitor printVisitor = new PrintVisitor();
            Visitor doVisitor = new DoVisitor();
            printVisitor.visitCar(car);
            doVisitor.visitCar(car);
        }
    }
```

The following example is an example in the C++ programming language:

```
#include <string>
#include <iostream>
#include <algorithm>
#include <list>

using namespace std;

class Wheel;
class Engine;
class Body;
class Car;

// interface to all car 'parts'
struct Visitor
{
  virtual void visit(Wheel& wheel) = 0;
  virtual void visit(Engine& engine) = 0;
  virtual void visit(Body& body) = 0;
  virtual void visitCar(Car& car) = 0;
};

// interface to one part
struct CarElement
{
  virtual void accept(Visitor &visitor) = 0;
};

// wheel element, there are four wheels with unique name
class Wheel : public CarElement
{
public:
    Wheel(const string &name)
    {
      _name = name;
    }
    string getName()
```

```
    {
      return _name;
    }
    void accept(Visitor& visitor)
    {
      visitor.visit(*this);
    }
private:
    string _name;
};

// engine
class Engine : public CarElement
{
public:
  void accept(Visitor& visitor)
  {
    visitor.visit(*this);
  }
};

// body
class Body : public CarElement
{
public:
  void accept(Visitor& visitor)
  {
    visitor.visit(*this);
  }
};

// car, all car elements(parts) together
class Car
{
public:
  list<CarElement*> & getElements()
  {
    return elements;
  }
  Car()
  {
    elements.push_back( new Wheel("front left") );
    elements.push_back( new Wheel("front right") );
    elements.push_back( new Wheel("back left") );
    elements.push_back( new Wheel("back right") );
    elements.push_back( new Body() );
    elements.push_back( new Engine() );
  }
  ~Car()
  {
    while ( !elements.empty() )
    {
      CarElement *p = *(elements.begin());
      delete p;
      elements.erase( elements.begin() );
    }
  }
private:
  list<CarElement*> elements;
};

// PrintVisitor and DoVisitor show by using a different implementation the struct 'car' is unchanged
// even though the algorithm are different in PrintVisitor and DoVisitor.
class PrintVisitor : public Visitor
{
public:
  void visit(Wheel& wheel)
  {
    cout << "Visiting " << wheel.getName() << " wheel" << endl;
  }
  void visit(Engine& engine)
```

```
  {
    cout << "Visiting engine" << endl;
  }
  void visit(Body& body)
  {
    cout << "Visiting body" << endl;
  }
  void visitCar(Car& car)
  {
    cout << endl << "Visiting car" << endl;
    list<CarElement*> & lst = car.getElements();
    list<CarElement*>::iterator it = lst.begin();
    while ( it != lst.end() )
    {
      (*it)->accept(*this);     // this issues the callback i.e. to this from the element
      ++it;
    }
    cout << "Visited car" << endl;
  }
};

class DoVisitor : public Visitor
{
public:
  // these are specific implementations added to the original object without modifying the original struct
  void visit(Wheel& wheel)
  {
    cout << "Kicking my " << wheel.getName() << endl;
  }
  void visit(Engine& engine)
  {
    cout << "Starting my engine" << endl;
  }
  void visit(Body& body)
  {
    cout << "Moving my body" << endl;
  }
  void visitCar(Car& car)
  {
    cout << endl << "Starting my car" << endl;
    list<CarElement*> & lst = car.getElements();
    list<CarElement*>::iterator it = lst.begin();
    while ( it != lst.end() )
    {
      (*it)->accept(*this);     // this issues the callback i.e. to this from the element
      ++it;
    }
    cout << "Stopped car" << endl;
  }
};

int main(int argc, char* argv[])
{
  Car car;
  auto_ptr<Visitor> printVisitor( new PrintVisitor );
  auto_ptr<Visitor> doVisitor( new DoVisitor );
  printVisitor->visitCar(car);
  doVisitor->visitCar(car);
  return 0;
}
```

State

Aside from potentially improving separation of concerns, the visitor pattern has an additional advantage over simply calling a polymorphic method: a visitor object can have state. This is extremely useful in many cases where the action performed on the object depends on previous such actions.

An example of this is a pretty-printer in a programming language implementation (such as a compiler or interpreter). Such a pretty-printer object (implemented as a visitor, in this example), will visit nodes in a data structure that represents a parsed and processed program. The pretty-printer will then generate a textual representation of the program tree. In order to make the representation human readable, the pretty-printer should properly indent program statements and expressions. The *current indentation level* can then be tracked by the visitor as its state, correctly applying encapsulation, whereas in a simple polymorphic method invocation, the indentation level would have to be exposed as a parameter and the caller would rely on the method implementation to use and propagate this parameter correctly.