Computer Science Design Patterns

en.wikibooks.org

December 29, 2013

Contents

1 Computer Science Design Patterns/Print version

2 Abstract Factory

The *abstract factory* pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client does not know (or care) 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 their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface. An example of this would be an abstract factory class `DocumentCreator` that provides interfaces to create a number of products (e.g. `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 instance 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 only need to know how to handle the abstract `Letter` or `Resume` class, not the specific version that it got from the concrete factory. A *factory* is the location of a concrete class in the code at which objects are constructed. The intent in employing the pattern is to insulate the creation of objects from their usage and to create families of related objects without having to depend on their concrete classes. 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 implementations 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. Additionally, higher levels of separation and abstraction can result in systems which are more difficult to debug and maintain. Therefore, as in all software designs, the trade-offs must be carefully evaluated.

2.1 Definition

The essence of the Abstract Factory Pattern is to "Provide an interface for creating families of related or dependent objects without specifying their concrete classes".

2.2 Usage

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 (that requested the object from the factory) does not know — and is not burdened by — the actual concrete type of the object that 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 related to it. 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 that 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.

2.3 Structure

2.3.1 Class diagram

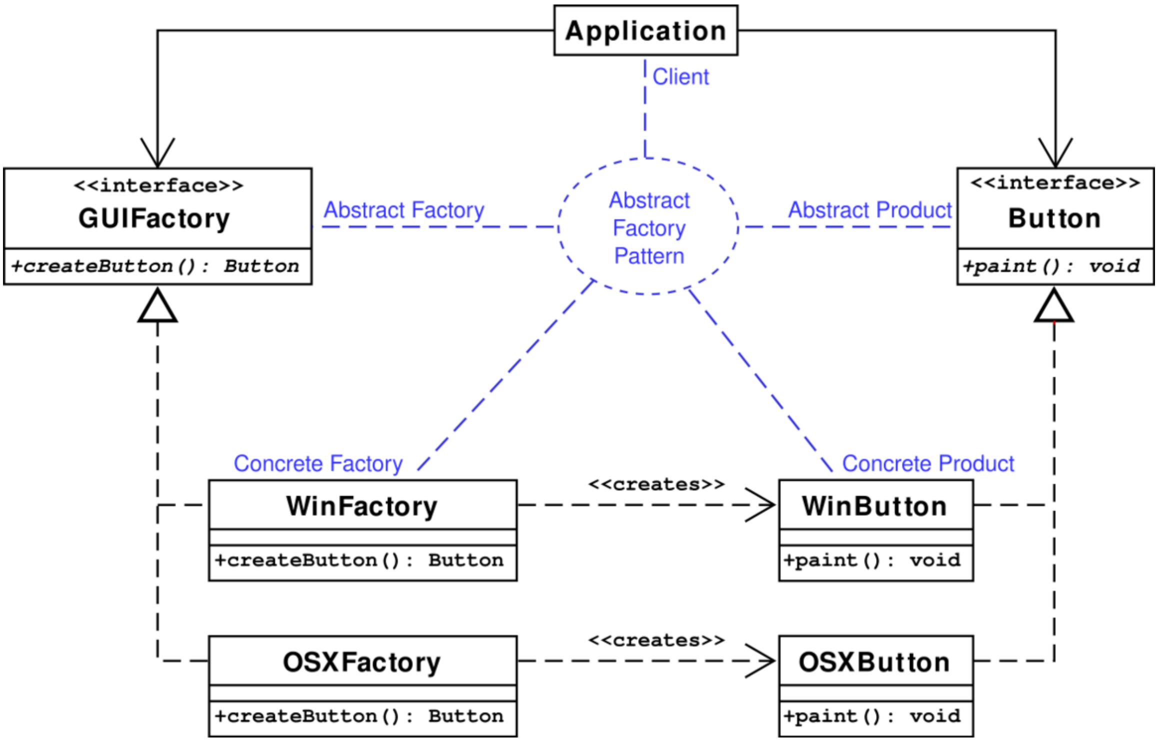


Figure 1

The method `createButton` on the `GuiFactory` interface returns objects of type `Button`. What implementation of `Button` is returned depends on which implementation of `GuiFactory` is handling the method call. Note that, for conciseness, this class diagram only shows the class relations for creating one type of object.

2.3.2 Lepus3 chart (legend)

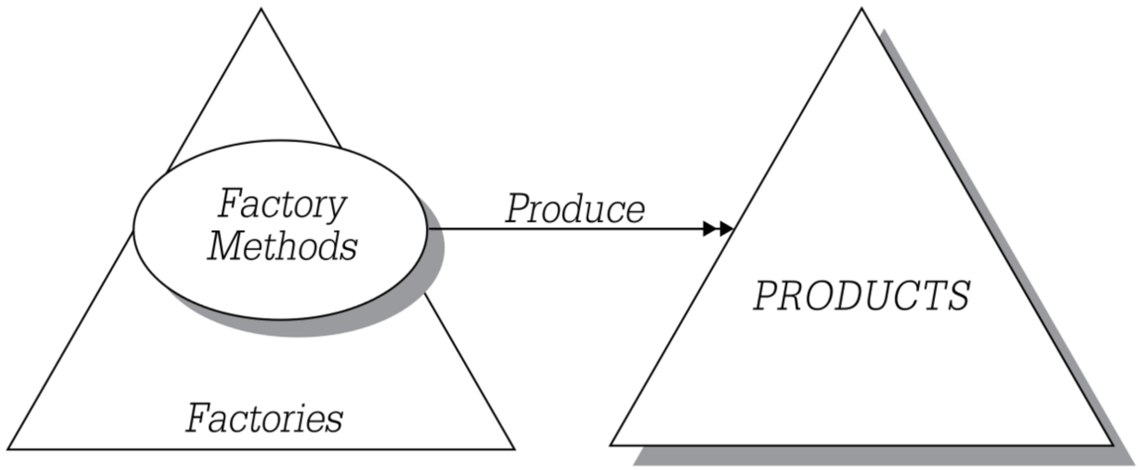


Figure 2

2.3.3 UML diagram

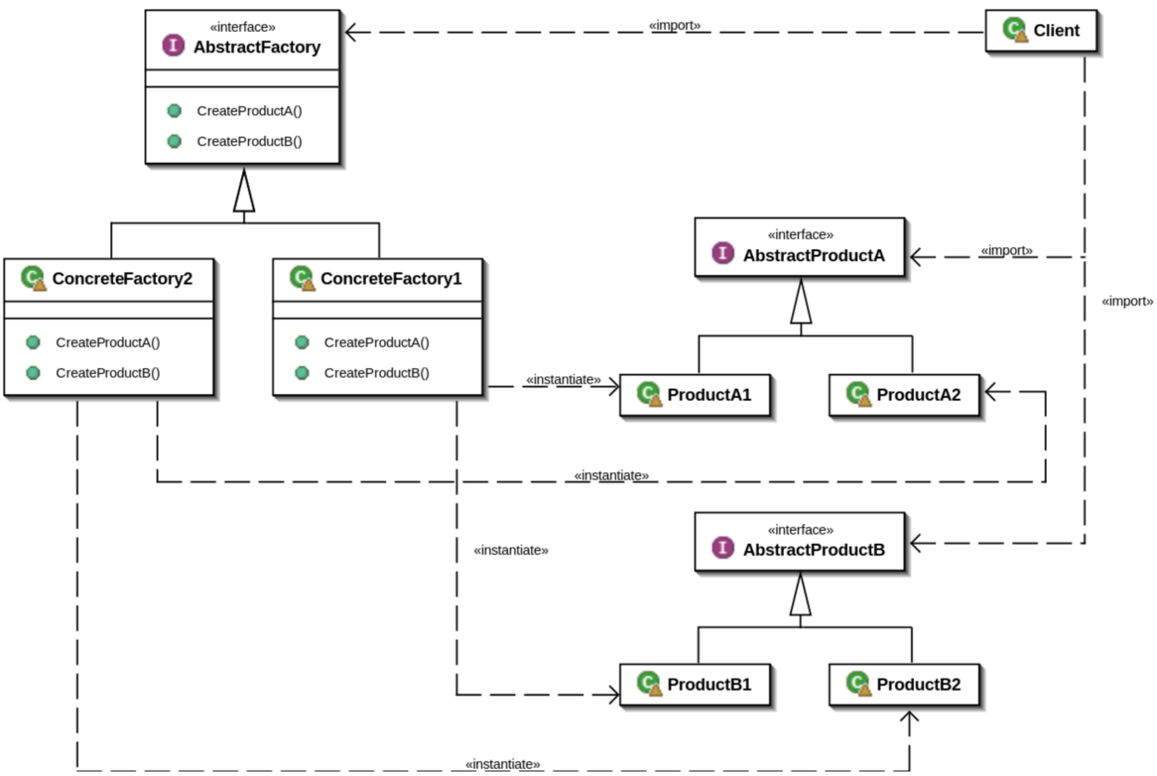


Figure 3

2.4 Implementations

The output should be either "I'm a WinButton" or "I'm an OSXButton" depending on which kind of factory was used. Note that the Application has no idea what kind of GUIFactory it is given or even what kind of Button that factory creates.

Implementation in C#

```
/* GUIFactory example -- */

using System;
using System.Configuration;

namespace AbstractFactory {
    public interface IButton {
        void Paint();
    }

    public interface IGUIFactory {
        IButton CreateButton();
    }

    public class OSXButton : IButton { // Executes fourth if OS:OSX
        public void Paint() {
            System.Console.WriteLine("I'm an OSXButton");
        }
    }

    public class WinButton : IButton { // Executes fourth if OS:WIN
        public void Paint() {
            System.Console.WriteLine("I'm a WinButton");
        }
    }

    public class OSXFactory : IGUIFactory { // Executes third if OS:OSX
        IButton IGUIFactory.CreateButton() {
            return new OSXButton();
        }
    }

    public class WinFactory : IGUIFactory { // Executes third if OS:WIN
        IButton IGUIFactory.CreateButton() {
            return new WinButton();
        }
    }

    public class Application {
        public Application(IGUIFactory factory) {
            IButton button = factory.CreateButton();
            button.Paint();
        }
    }

    public class ApplicationRunner {
        static IGUIFactory CreateOsSpecificFactory() { // Executes second
            // Contents of App.Config associated with this C# project
            //?xml version="1.0" encoding="utf-8" ?
            //configuration
            //  appSettings
            //    !-- Uncomment either Win or OSX OS_TYPE to test --
            //    add key="OS_TYPE" value="Win" /
            //    !-- add key="OS_TYPE" value="OSX" / --
            //  /appSettings
            ///configuration
```

```
            string sysType = ConfigurationSettings.AppSettings["OS_TYPE"];
            if (sysType == "Win") {
                return new WinFactory();
            } else {
                return new OSXFactory();
            }
        }

        static void Main() { // Executes first
            new Application(CreateOsSpecificFactory());
            Console.ReadLine();
        }
    }
}
```

Implementation in C++

```
/* GUIFactory example */

#include iostream

class Button {
public:
    virtual void paint() = 0;
    virtual ~Button() { }
};

class WinButton : public Button {
public:
    void paint() {
        std::cout  "I'm a WinButton";
    }
};

class OSXButton : public Button {
public:
    void paint() {
        std::cout  "I'm an OSXButton";
    }
};

class GUIFactory {
public:
    virtual Button* createButton() = 0;
    virtual ~GUIFactory() { }
};

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

    ~WinFactory() { }
};

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

    ~OSXFactory() { }
};

class Application {
```

```
public:
    Application(GUIFactory* factory) {
        Button* button = factory-createButton();
        button-paint();
        delete button;
        delete factory;
    }
};

GUIFactory* createOsSpecificFactory() {
    int sys;
    std::cout  "Enter OS type (0: Windows, 1: MacOS X): ";
    std::cin  sys;

    if (sys == 0) {
        return new WinFactory();
    } else {
        return new OSXFactory();
    }
}

int main() {
    Application application(createOsSpecificFactory());

    return 0;
}
```

Implementation in Java

```
/* GUIFactory example -- */

interface Button {
    void paint();
}

interface GUIFactory {
    Button createButton();
}

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

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

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

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

class Application {
    public Application(GUIFactory factory) {
        Button button = factory.createButton();
```

```
        button.paint();
    }
}

public class ApplicationRunner {
    public static void main(String[] args) {
        new Application(createOsSpecificFactory());
    }

    public static GUIFactory createOsSpecificFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) return new WinFactory();
        else return new OSXFactory();
    }
}
```

Implementation in Objective-C

```
/* GUIFactory example -- */

#import Foundation/Foundation.h

@protocol Button NSObject
- (void)paint;
@end

@interface WinButton : NSObject Button
@end

@interface OSXButton : NSObject Button
@end

@protocol GUIFactory
- (idButton)createButton;
@end

@interface WinFactory : NSObject GUIFactory
@end

@interface OSXFactory : NSObject GUIFactory
@end

@interface Application : NSObject
- (id)initWithGUIFactory:(id)factory;
+ (id)createOsSpecificFactory:(int)type;
@end

@implementation WinButton
- (void)paint {
    NSLog(@"I am a WinButton.");
}
@end

@implementation OSXButton
- (void)paint {
    NSLog(@"I am a OSXButton.");
}
@end

@implementation WinFactory
- (idButton)createButton {
    return [[[WinButton alloc] init] autorelease];
}
@end

@implementation OSXFactory
```

```
- (idButton)createButton {
    return [[[OSXButton alloc] init] autorelease];
}
@end

@implementation Application
- (id)initWithGUIFactory:(idGUIFactory)factory {
    if (self = [super init]) {
        id button = [factory createButton];
        [button paint];
    }
    return self;
}
+ (idGUIFactory)createOsSpecificFactory:(int)type {
    if (type == 0) {
        return [[[WinFactory alloc] init] autorelease];
    } else {
        return [[[OSXFactory alloc] init] autorelease];
    }
}
@end

int main(int argc, char* argv[]) {
    @autoreleasepool {
        [[Application alloc] initWithGUIFactory:[Application
 createOsSpecificFactory:0]];// 0 is WinButton
    }
    return 0;
}
```

Implementation in Lua

```
--[[
    Because Lua is a highly dynamic Language, an OOP scheme is implemented by
 the programmer.
    The OOP scheme implemented here implements interfaces using documentation.


    A Factory Supports:
     - factory:CreateButton()

    A Button Supports:
     - button:Paint()
]]

-- Create the OSXButton Class
do
    OSXButton = {}
    local mt = { __index = OSXButton }

    function OSXButton:new()
        local inst = {}
        setmetatable(inst, mt)
        return inst
    end

    function OSXButton:Paint()
        print("I'm a fancy OSX button!")
    end
end

-- Create the WinButton Class
do
    WinButton = {}
    local mt = { __index = WinButton }
```

```
    function WinButton:new()
        local inst = {}
        setmetatable(inst, mt)
        return inst
    end

    function WinButton:Paint()
        print("I'm a fancy Windows button!")
    end
end

-- Create the OSXGuiFactory Class
do
    OSXGuiFactory = {}
    local mt = { __index = OSXGuiFactory }

    function OSXGuiFactory:new()
        local inst = {}
        setmetatable(inst, mt)
        return inst
    end

    function OSXGuiFactory:CreateButton()
        return OSXButton:new()
    end
end

-- Create the WinGuiFactory Class
do
    WinGuiFactory = {}
    local mt = { __index = WinGuiFactory }

    function WinGuiFactory:new()
        local inst = {}
        setmetatable(inst, mt)
        return inst
    end

    function WinGuiFactory:CreateButton()
        return WinButton:new()
    end
end

-- Table to keep track of what GuiFactories are available
GuiFactories = {
    ["Win"] = WinGuiFactory,
    ["OSX"] = OSXGuiFactory,
}

--[[ Inside an OS config script ]]
OS_VERSION = "Win"

--[[ Using the Abstract Factory in some the application script ]]

-- Selecting the factory based on OS version
MyGuiFactory = GuiFactories[OS_VERSION]:new()

-- Using the factory
osButton = MyGuiFactory:CreateButton()
osButton:Paint()
```

Implementation in PHP

This abstract factory creates books.

```
/*
```

```
 * BookFactory classes
*/
abstract class AbstractBookFactory {
    abstract function makePHPBook();
    abstract function makeMySQLBook();
}

class OReillyBookFactory extends AbstractBookFactory {
    private $context = "OReilly";
    function makePHPBook() {
        return new OReillyPHPBook;
    }
    function makeMySQLBook() {
        return new OReillyMySQLBook;
    }
}

class SamsBookFactory extends AbstractBookFactory {
    private $context = "Sams";
    function makePHPBook() {
        return new SamsPHPBook;
    }
    function makeMySQLBook() {
        return new SamsMySQLBook;
    }
}

/*
 *   Book classes
*/
abstract class AbstractBook {
    abstract function getAuthor();
    abstract function getTitle();
}

abstract class AbstractMySQLBook {
    private $subject = "MySQL";
}

class OReillyMySQLBook extends AbstractMySQLBook {
    private $author;
    private $title;
    function __construct() {
        $this-author = 'George Reese, Randy Jay Yarger, and Tim King';
        $this-title = 'Managing and Using MySQL';
    }
    function getAuthor() {
        return $this-author;
    }
    function getTitle() {
        return $this-title;
    }
}

class SamsMySQLBook extends AbstractMySQLBook {
    private $author;
    private $title;
    function __construct() {
        $this-author = 'Paul Dubois';
        $this-title = 'MySQL, 3rd Edition';
    }
    function getAuthor() {
        return $this-author;
    }
    function getTitle() {
        return $this-title;
    }
}
```

```
abstract class AbstractPHPBook {
    private $subject = "PHP";
}

class OReillyPHPBook extends AbstractPHPBook {
    private $author;
    private $title;
    private static $oddOrEven = 'odd';
    function __construct() {
        //alternate between 2 books
        if ('odd' == self::$oddOrEven) {
            $this-author = 'Rasmus Lerdorf and Kevin Tatroe';
            $this-title = 'Programming PHP';
            self::$oddOrEven = 'even';
        } else {
            $this-author = 'David Sklar and Adam Trachtenberg';
            $this-title = 'PHP Cookbook';
            self::$oddOrEven = 'odd';
        }
    }
    function getAuthor() {
        return $this-author;
    }
    function getTitle() {
        return $this-title;
    }
}

class SamsPHPBook extends AbstractPHPBook {
    private $author;
    private $title;
    function __construct() {
        //alternate randomly between 2 books
        mt_srand((double)microtime() * 10000000);
        $rand_num = mt_rand(0, 1);

        if (1  $rand_num) {
            $this-author = 'George Schlossnagle';
            $this-title = 'Advanced PHP Programming';
        }
        else {
            $this-author = 'Christian Wenz';
            $this-title = 'PHP Phrasebook';
        }
    }
    function getAuthor() {
        return $this-author;
    }
    function getTitle() {
        return $this-title;
    }
}

/*
 *   Initialization
*/

writeln('BEGIN TESTING ABSTRACT FACTORY PATTERN');
writeln('');

writeln('testing OReillyBookFactory');
$bookFactoryInstance = new OReillyBookFactory;
testConcreteFactory($bookFactoryInstance);
writeln('');

writeln('testing SamsBookFactory');
$bookFactoryInstance = new SamsBookFactory;
```

```
testConcreteFactory($bookFactoryInstance);

writeln("END TESTING ABSTRACT FACTORY PATTERN");
writeln('');

function testConcreteFactory($bookFactoryInstance) {
    $phpBookOne = $bookFactoryInstance-makePHPBook();
    writeln('first php Author: '.$phpBookOne-getAuthor());
    writeln('first php Title: '.$phpBookOne-getTitle());

    $phpBookTwo = $bookFactoryInstance-makePHPBook();
    writeln('second php Author: '.$phpBookTwo-getAuthor());
    writeln('second php Title: '.$phpBookTwo-getTitle());

    $mySqlBook = $bookFactoryInstance-makeMySQLBook();
    writeln('MySQL Author: '.$mySqlBook-getAuthor());
    writeln('MySQL Title: '.$mySqlBook-getTitle());
}

function writeln($line_in) {
    echo $line_in."br/";
}
```

3 Builder

The builder pattern is useful to avoid a huge list of constructors for a class. Let's imagine a class that store the mode of transport (of an employee for instance). Here is the constructor that takes a MetroLine object as parameter:

```
modeOfTransport(MetroLine)
```

Now we need a constructor for the one that uses the car, the bus or the train:

```
new modeOfTransport(MetroLine)
new modeOfTransport()
new modeOfTransport(BusLine)
new modeOfTransport(Train)
```

For some of them, we need to indicate a travel allowance:

```
new modeOfTransport(MetroLine)
new modeOfTransport(MetroLine, Integer)
new modeOfTransport()
new modeOfTransport(Integer)
new modeOfTransport(BusLine)
new modeOfTransport(BusLine, Integer)
new modeOfTransport(Train)
new modeOfTransport(Train, Integer)
```

We have employees that have several modes of transport to go to work. We realize that the list of constructors is coming to a mess. Each new parameter leads to an exponent duplication of constructors. If some parameters have the same type, it becomes very confusing. A solution to this issue would be to first build a fake object and then fill it calling methods on it:

```
new modeOfTransport(MetroLine)
modeOfTransport.setMetroLine(MetroLine)
modeOfTransport.setCarTravel()
modeOfTransport.setBusLine(BusLine)
modeOfTransport.setTrain(Train)
modeOfTransport.setTravelAllowance(Integer)
```

The list of methods is no more exponent but the state of the object may be inconsistent. A better solution would be to set all the parameters to an object of another class, a pre-constructor, and then pass this object to the constructor of ModeOfTransport:

```
modeOfTransportPreConstructor.setMetroLine(MetroLine)
modeOfTransportPreConstructor.setCarTravel()
modeOfTransportPreConstructor.setBusLine(BusLine)
modeOfTransportPreConstructor.setTrain(Train)
modeOfTransportPreConstructor.setTravelAllowance(Integer)
new modeOfTransport(ModeOfTransportPreConstructor)
```

This solution is even better. We only have a valid ModeOfTransport object. However, the ModeOfTransport constructor can be called with a half-filled pre-constructor. So the pre-constructor should be a *builder* and should have a method that returns the ModeOfTransport object. This method will only return an object if the builder has been correctly filled, otherwise it returns null:

```
modeOfTransportBuilder.setMetroLine(MetroLine)
modeOfTransportBuilder.setCarTravel()
modeOfTransportBuilder.setBusLine(BusLine)
modeOfTransportBuilder.setTrain(Train)
modeOfTransportBuilder.setTravelAllowance(Integer)
modeOfTransport := modeOfTransportBuilder.getResult()
```

So the solution is to use a builder class. Let's see the structure of the code on the UML class diagram:

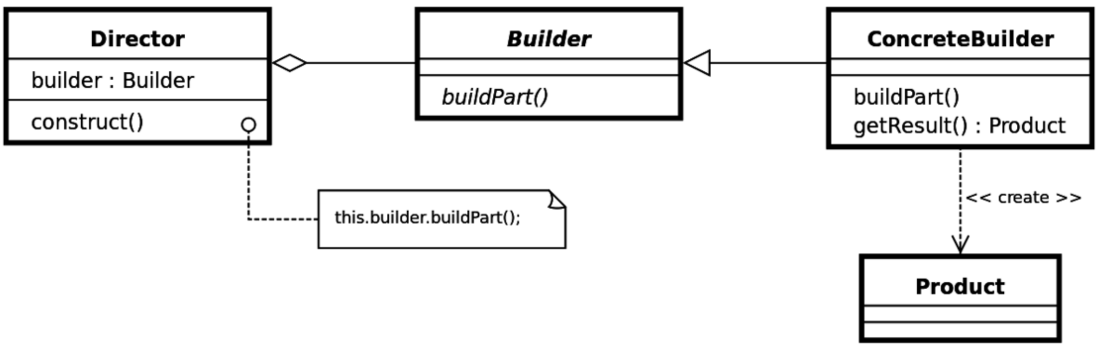


Figure 4 Builder Structure

- ***Builder***: Abstract interface for creating objects (product).
- **Concrete Builder**: Provides implementation for Builder. It is an object able to construct other objects. Constructs and assembles parts to build the objects.

The builder pattern can be used for objects that contain flat data (html code, SQL query, X.509 certificate...), that is to say data that can't be easily edited. This type of data can not be edited step by step and must be edited at once. The best way to construct such an object is to use a builder class.

3.1 Examples

In Java, the `StringBuffer` and `StringBuilder` classes follow the builder design pattern. They are used to build `String` objects.

3.2 Cost

You can easily decide to use it. At worst, the pattern is useless.

3.2.1 Creation

Starting from a plain old class with a public constructor, implementing the design pattern is not very expensive. You can create the builder class aside your code. Then you will have to remove the existing constructor calls to use the builder instead. The refactoring is hardly automatic.

3.2.2 Maintenance

This design pattern has only small drawbacks. It may lead to small redundancies between the class and the builder structures but the both class have usually different structures. However, it is expensive to evolve to an abstract factory.

3.2.3 Removal

The refactoring is hardly automatic. It should be done manually.

3.3 Advices

- Put the *builder* term in the name of the builder class to indicate the use of the pattern to the other developpers.
- Build your builder class as a fluent interface. It would increase the pattern interest.
- If the target class contains flat data, your builder class can be constructed as a Composite[1] that implements the Interpreter[2] pattern.

1 http://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Composite
2 http://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Interpreter

3.4 Implementations

Implementation in Java

Building a car.

```
/**

 * Can have GPS, trip computer and a various number of seaters. Can be a city
 car, a sport car or a cabriolet.

 */

public class Car {

  /**

   * The description of the car.

   */

  private String description = null;



  /**

   * Construct and return the car.

   * @param aDescription The description of the car.

   */

  public Car(String aDescription) {
```

```
      description = aDescription;
    }

  /**
   * Print the car.
   * @return A flat representation.
   */
  public String toString() {
      return description;
    }
}

/**
 *
 */
public class CarBuilder {
  /**
```

```
 * Sport car.

 */

private static final int SPORT_CAR = 1;



/**

 * City car.

 */

private static final int CITY_CAR = 2;



/**

 * Cabriolet.

 */

private static final int CABRIOLET = 3;



/**

 * The type of the car.

 */
```

```
private int carType;



/**

 * True if the car has a trip computer.

 */

private boolean hasTripComputer;



/**

 * True if the car has a GPS.

 */

private boolean hasGPS;



/**

 * The number of seaters the car may have.

 */

private int seaterNumber;
```

```
/**

 * Construct and return the car.

 * @return a Car with the right options.

 */

public Car getResult() {

    return new Car((carType == CITY_CAR) ? "A city car" : ((carType ==
SPORT_CAR) ? "A sport car" : "A cabriolet")

      + " with " + seaterNumber + " seaters"

      + (hasTripComputer ? " with a trip computer" : "")

      + (hasGPS ? " with a GPS" : "")

      + ".");

  }



/**

 * Tell the builder the number of seaters.

 * @param number the number of seaters the car may have.

 */

public void setSeaters(int number) {
```

```
    seaterNumber = number;

  }




/**

 * Make the builder remember that the car is a city car.

 */

public void setCityCar() {

    carType = CITY_CAR;

  }




/**

 * Make the builder remember that the car is a cabriolet.

 */

public void setCabriolet() {

    carType = CABRIOLET;

  }
```

```
/**
 * Make the builder remember that the car is a sport car.
 */
public void setSportCar() {
    carType = SPORT_CAR;
  }


/**
 * Make the builder remember that the car has a trip computer.
 */
public void setTripComputer() {
    hasTripComputer = true;
  }


/**
 * Make the builder remember that the car has not a trip computer.
 */
```

```
public void unsetTripComputer() {
    hasTripComputer = false;
  }

/**
 * Make the builder remember that the car has a global positioning system.
 */
public void setGPS() {
    hasGPS = true;
  }

/**
 * Make the builder remember that the car has not a global positioning system.
 */
public void unsetGPS() {
    hasGPS = false;
  }
```

```
}

/**
 * Construct a CarBuilder called carBuilder and build a car.
 */
public class Director {
  public static void main(String[] args) {
    CarBuilder carBuilder = new CarBuilder();
    carBuilder.setSeaters(2);
    carBuilder.setSportCar();
    carBuilder.setTripComputer();
    carBuilder.unsetGPS();
    Car car = carBuilder.getResult();

    System.out.println(car);
  }
}
```

It will produce:

```
A sport car with 2 seaters with a trip computer.
```

Building a pizza.

```
 /** "Product" */

class Pizza {

 private String dough = "";

 private String sauce = "";

 private String topping = "";



 public void setDough(final String dough) {

  this.dough = dough;

 }



 public void setSauce(final String sauce) {

  this.sauce = sauce;

 }
```

```
 public void setTopping(final String topping) {
  this.topping = topping;
 }
}

 /** "Abstract Builder" */
abstract class PizzaBuilder {
 protected Pizza pizza;

 public abstract void buildDough();
 public abstract void buildSauce();
 public abstract void buildTopping();

 public void createNewPizzaProduct() {
  this.pizza = new Pizza();
 }
```

```
 public Pizza getPizza() {

  return this.pizza;

 }

}



 /** "ConcreteBuilder" */

class HawaiianPizzaBuilder extends PizzaBuilder {

 @Override public void buildDough() {

  this.pizza.setDough("cross");

 }



 @Override public void buildSauce() {

  this.pizza.setSauce("mild");

 }



 @Override public void buildTopping() {

  this.pizza.setTopping("ham+pineapple");
```

```
 }
}

 /** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
 @Override public void buildDough() {
  this.pizza.setDough("pan baked");
 }

 @Override public void buildSauce() {
  this.pizza.setSauce("hot");
 }

 @Override public void buildTopping() {
  this.pizza.setTopping("pepperoni+salami");
 }
}
```

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

 public void setPizzaBuilder(final PizzaBuilder pb) {
  this.pizzaBuilder = pb;
 }

 public Pizza getPizza() {
  return this.pizzaBuilder.getPizza();
 }

 public void constructPizza() {
  this.pizzaBuilder.createNewPizzaProduct();
  this.pizzaBuilder.buildDough();
  this.pizzaBuilder.buildSauce();
```

```
  this.pizzaBuilder.buildTopping();
 }
}

 /** A customer ordering a pizza. */
class BuilderExample {

 public static void main(final String[] args) {

  final Waiter waiter = new Waiter();

  final PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
  final PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

  waiter.setPizzaBuilder(hawaiianPizzaBuilder);
  waiter.constructPizza();
```

```
  final Pizza pizza = waiter.getPizza();

  waiter.setPizzaBuilder(spicyPizzaBuilder);

  waiter.constructPizza();

  final Pizza anotherPizza = waiter.getPizza();

 }

}
```

Implementation in C#

```
using System;

namespace BuilderPattern
{
  // Builder - abstract interface for creating objects (the product, in this
 case)
  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();
  }
  // Concrete Builder - provides implementation for Builder; an object able to
 construct other objects.
  // Constructs and assembles parts to build the objects
  class HawaiianPizzaBuilder : PizzaBuilder
  {
```

```
    public override void BuildDough()
    {
      pizza.dough = "cross";
    }

    public override void BuildSauce()
    {
      pizza.sauce = "mild";
    }

    public override void BuildTopping()
    {
      pizza.topping = "ham+pineapple";
    }
  }
  // Concrete Builder - provides implementation for Builder; an object able to
 construct other objects.
  // Constructs and assembles parts to build the objects
  class SpicyPizzaBuilder : PizzaBuilder
  {
    public override void BuildDough()
    {
      pizza.dough = "pan baked";
    }

    public override void BuildSauce()
    {
      pizza.sauce = "hot";
    }

    public override void BuildTopping()
    {
      pizza.topping = "pepperoni + salami";
    }
  }

  // Director - responsible for managing the correct sequence of object
 creation.
  // Receives a Concrete Builder as a parameter and executes the necessary
 operations on it.
  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();
    }
  }

  // Product - The final object that will be created by the Director using
 Builder
  public class Pizza
  {
    public string dough = string.Empty;
```

```
    public string sauce = string.Empty;
    public string topping = string.Empty;
  }

  class Program
  {
    static void Main(string[] args)
    {
      PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
      Cook cook = new Cook();
      cook.SetPizzaBuilder(hawaiianPizzaBuilder);
      cook.ConstructPizza();
      // create the product
      Pizza hawaiian = cook.GetPizza();

      PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();
      cook.SetPizzaBuilder(spicyPizzaBuilder);
      cook.ConstructPizza();
      // create another product
      Pizza spicy = cook.GetPizza();
    }
  }

}
```

Implementation in C++

```
#include string
#include iostream
using namespace std;

// "Product"
class Pizza {
public:
        void dough(const string dough) {
                dough_ = dough;
        }

        void sauce(const string sauce) {
                sauce_ = sauce;
        }

        void topping(const string topping) {
                topping_ = topping;
        }

        void open() const {
                cout  "Pizza with "  dough_  " dough, "  sauce_  " sauce and "
                             topping_  " topping. Mmm."  endl;
        }

private:
        string dough_;
        string sauce_;
        string topping_;
};

// "Abstract Builder"
class PizzaBuilder {
public:
       // Chinmay Mandal : This default constructor may not be required here
        PizzaBuilder()
        {
           // Chinmay Mandal : Wrong syntax
           // pizza_ = new Pizza;
        }
```

```
        const Pizza pizza() {
                return pizza_;
        }

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

protected:
        Pizza pizza_;
};

class HawaiianPizzaBuilder : public PizzaBuilder {
public:
        void buildDough() {
                pizza_.dough("cross");
        }

        void buildSauce() {
                pizza_.sauce("mild");
        }

        void buildTopping() {
                pizza_.topping("ham+pineapple");
        }
};

class SpicyPizzaBuilder : public PizzaBuilder {
public:
        void buildDough() {
                pizza_.dough("pan baked");
        }

        void buildSauce() {
                pizza_.sauce("hot");
        }

        void buildTopping() {
                pizza_.topping("pepperoni+salami");
        }
};

class Cook {
public:
        Cook()
                : pizzaBuilder_(NULL/*nullptr*/)//Chinmay Mandal : nullptr
 replaced with NULL.
        {       }

        ~Cook() {
                if (pizzaBuilder_)
                        delete pizzaBuilder_;
        }

        void pizzaBuilder(PizzaBuilder* pizzaBuilder) {
                if (pizzaBuilder_)
                        delete pizzaBuilder_;

                pizzaBuilder_ = pizzaBuilder;
        }

        const Pizza getPizza() {
                return pizzaBuilder_-pizza();
        }

        void constructPizza() {
                pizzaBuilder_-buildDough();
                pizzaBuilder_-buildSauce();
```

```
                pizzaBuilder_-buildTopping();
        }

private:
        PizzaBuilder* pizzaBuilder_;
};

int main() {
        Cook cook;
        cook.pizzaBuilder(new HawaiianPizzaBuilder);
        cook.constructPizza();

        Pizza hawaiian = cook.getPizza();
        hawaiian.open();

        cook.pizzaBuilder(new SpicyPizzaBuilder);
        cook.constructPizza();

        Pizza spicy = cook.getPizza();
        spicy.open();
}
```

Implementation in PHP

```
?php
/** "Product" */
class Pizza {

    protected $dough;
    protected $sauce;
    protected $topping;

    public function setDough($dough) {
        $this-dough = $dough;
    }

    public function setSauce($sauce) {
        $this-sauce = $sauce;
    }

    public function setTopping($topping) {
        $this-topping = $topping;
    }

    public function showIngredients() {
        echo "Dough   : ".$this-dough."br/";
        echo "Sauce   : ".$this-sauce."br/";
        echo "Topping : ".$this-topping."br/";
    }
}

/** "Abstract Builder" */
abstract class PizzaBuilder {

    protected $pizza;

    public function getPizza() {
        return $this-pizza;
    }

    public function createNewPizzaProduct() {
        $this-pizza = new Pizza();
    }

    public abstract function buildDough();
    public abstract function buildSauce();
```

```
    public abstract function buildTopping();
}

/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
    public function buildDough() {
        $this-pizza-setDough("cross");
    }

    public function buildSauce() {
        $this-pizza-setSauce("mild");
    }

    public function buildTopping() {
        $this-pizza-setTopping("ham + pineapple");
    }
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {

    public function buildDough() {
        $this-pizza-setDough("pan baked");
    }

    public function buildSauce() {
        $this-pizza-setSauce("hot");
    }

    public function buildTopping() {
        $this-pizza-setTopping("pepperoni + salami");
    }
}

/** "Director" */
class Waiter {

    protected $pizzaBuilder;

    public function setPizzaBuilder(PizzaBuilder $pizzaBuilder) {
        $this-pizzaBuilder = $pizzaBuilder;
    }

    public function getPizza() {
        return $this-pizzaBuilder-getPizza();
    }

    public function constructPizza() {
        $this-pizzaBuilder-createNewPizzaProduct();
        $this-pizzaBuilder-buildDough();
        $this-pizzaBuilder-buildSauce();
        $this-pizzaBuilder-buildTopping();
    }
}

class Tester {

    public static function main() {

        $oWaiter               = new Waiter();
        $oHawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        $oSpicyPizzaBuilder    = new SpicyPizzaBuilder();

        $oWaiter-setPizzaBuilder($oHawaiianPizzaBuilder);
        $oWaiter-constructPizza();

        $pizza = $oWaiter-getPizza();
        $pizza-showIngredients();
```

```
        echo "br/";

        $oWaiter-setPizzaBuilder($oSpicyPizzaBuilder);
        $oWaiter-constructPizza();

        $pizza = $oWaiter-getPizza();
        $pizza-showIngredients();
    }
}

Tester::main();
```

output :

```
Dough : cross
Sauce : mild
Topping : ham + pineapple



Dough : pan baked
Sauce : hot
Topping : pepperoni + salami
```

Implementation in Ruby

```
# Product
class Pizza
  attr_accessor :dough, :sauce, :topping
end

# Abstract Builder
class PizzaBuilder
  def get_pizza
    @pizza
  end

  def create_new_pizza_product
    @pizza = Pizza.new
  end

  def build_dough; end
  def build_sauce; end
  def build_topping; end

end

# ConcreteBuilder
class HawaiianPizzaBuilder  PizzaBuilder
  def build_dough
    @pizza.dough = 'cross'
  end

  def build_sauce
    @pizza.sauce = 'mild'
  end

  def build_topping
    @pizza.topping = 'ham+pineapple'
  end
end

# ConcreteBuilder
```

```
class SpicyPizzaBuilder  PizzaBuilder
  def build_dough
    @pizza.dough = 'pan baked'
  end

  def build_sauce
    @pizza.sauce = 'hot'
  end

  def build_topping
    @pizza.topping = 'pepperoni+salami'
  end
end

# Director
class Waiter
  attr_accessor :pizza_builder

  def get_pizza
    pizza_builder.get_pizza
  end

  def construct_pizza
    pizza_builder.create_new_pizza_product
    pizza_builder.build_dough
    pizza_builder.build_sauce
    pizza_builder.build_topping
  end
end

# A customer ordering a pizza.
class BuilderExample
  def main(args = [])
    waiter = Waiter.new
    hawaiian_pizza_builder = HawaiianPizzaBuilder.new
    spicy_pizza_builder = SpicyPizzaBuilder.new

    waiter.pizza_builder = hawaiian_pizza_builder
    waiter.construct_pizza

    pizza = waiter.get_pizza
  end
end

puts BuilderExample.new.main.inspect
```

4 Print version

The factory method design pattern handles the problem of creating objects (products) without specifying the exact class of object that will be created. 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.

4.1 Intro to the Factory Pattern

The factory pattern is a design pattern used to promote encapsulation of data representation. It's primary purpose is to provide a way for users to retrieve an instance with a known compile-time type, but whose runtime type may actually be different. In other words, a factory method that is supposed to return an instance of the class *Foo* may return an instance of the class *Foo*, or it may return an instance of the class *Bar*, so long as *Bar* inherits from *Foo*. The reason for this is that it strengthens the boundary between implementor and client, hiding the true representation of the data (see Abstraction Barrier) from the user, thereby allowing the implementor to change this representation at anytime without affecting the client, as long as the client facing interface doesn't change.

4.2 Basic Implementation of the Factory Pattern

The general template for implementing the factory pattern is to provide a primary user facing class with static methods which the user can use to get instances with that type. Constructors are then made private/protected from the user, forcing them to use the static factory methods to get objects. The following Java[1] code shows a very simple implementation of the factory pattern for type *Foo*.

```
public class Foo {

  // Static factory method
  public static Foo getInstance() {
    // Inside this class, we have access to private methods
    return new Foo();
  }

  // Guarded constructor, only accessible from within this class, since it's
 marked private
  private Foo() {
    // Typical initialization code goes here
  }
```

1 http://en.wikibooks.org/wiki/Java

```
} // End class Foo
```

With this code, it would be impossible for a client of the code to use the **new** operator to get an instance of the class, as is traditionally done:

```
// Client code
Foo f = new Foo(); // Won't Work!
```

because the constructor is marked *private*. Instead, the client would have to use the factory method:

```
// Client code
Foo f = Foo.getInstance(); // Works!
```

It should be noted that even within a programming language community, there is no general consensus as to the naming convention of a factory method. Some suggest naming the method with the name of the class, similar to a normal constructor, but starting with a lowercase. Others say that this is confusing, and suggest using an accessor type syntax, like the *getInstance* style used above, though others complain that this may incorrectly imply a singleton implementation. Likewise, some offer *newInstance*, but this is criticized as being misleading in certain situations where a strictly *new* instance may not actually be returned (once again, refer to the singleton pattern). As such, we will not attempt to follow any particularly rigid standard here, we will simply try to use a name that makes the most sense for our current purposes.

4.3 Factory Pattern Implementation of the Alphabet

That's great, you know how to implement a real simple factory pattern, but what good did it do you? Users are asking for something that fits into type *Foo*, and they're getting an instance of the class *Foo*, how is that different from just calling the constructor? Well it's not, except that you're putting another function call on the stack (which is a bad thing). But that's only for the above case. We'll now discuss a more useful use of the factory pattern. Consider a simple type called *Letter*, representing a letter in the alphabet, which has the following client facing interface (i.e., public instance methods):

```
char toCharacter();
boolean isVowel();
boolean isConsonant();
```

We could implement this easily enough without using the factory method, which might start out something like this:

```
public class Letter {

  private char fTheLetter;

  public Letter(char aTheLetter) {
    fTheLetter = aTheLetter;
```

```
  }

  public char toCharacter() {
    return fTheLetter;
  }

  public boolean isVowel() {
    //TODO: we haven't implemented this yet
    return true;
  }


  public boolean isConsonant() {
    // TODO: we haven't implemented this yet
    return false;
  }

} // End class Letter
```

Fairly simple, but notice we haven't implemented the last two methods yet. We can still do it pretty easily. The first might look like this:

```
  public boolean isVowel() {
    return
      fTheLetter == 'a' ||
      fTheLetter == 'e' ||
      fTheLetter == 'i' ||
      fTheLetter == 'o' ||
      fTheLetter == 'u' ||
      fTheLetter == 'A' ||
      fTheLetter == 'E' ||
      fTheLetter == 'I' ||
      fTheLetter == 'O' ||
      fTheLetter == 'U';
  }
```

Now that's not *too* bad, but we still need to do *isConsonant*. Fortunately, we at least know in this case that if it's a vowel, it's not a consonant, and vice versa, so our last method could simply be:

```
  public boolean isConsonant() {
    return !this.isVowel();
  }
```

So what's the problem here? Basically, every time you call either of these methods, your program has to do all that checking. Granted, this isn't a real heavy burden for the Java Runtime Environment, but you can imagine a much more complex, much more time consuming operation. Wouldn't it be great if we could avoid doing this every time we call the method? Let's say, for instance, we could do it once when we create the object, and then not have to do it again. Well sure, we can do that. Here's an implementation that'll do that for us, and we still don't have to use the factory method:

```
public class Letter {

  private char fTheLetter;
  private boolean fIsVowel;

  public Letter(char aTheLetter) {
```

```
    fTheLetter = aTheLetter;
    fIsVowel = fTheLetter == 'a' ||
      fTheLetter == 'e' ||
      fTheLetter == 'i' ||
      fTheLetter == 'o' ||
      fTheLetter == 'u' ||
      fTheLetter == 'A' ||
      fTheLetter == 'E' ||
      fTheLetter == 'I' ||
      fTheLetter == 'O' ||
      fTheLetter == 'U';
  }

  public char toCharacter() {
    return fTheLetter;
  }

  public boolean isVowel() {
    return fIsVowel;
  }


  public boolean isConsonant() {
    return !fIsVowel;
  }

} // End class Letter
```

Notice how we moved the lengthy operation into the constructor, and stored the result. OK, so now we're all fine and dandy, no? Sure, but let's say you came up with a new idea, a different implementation: you want to split this type into two classes, one class to handle the vowels, and one to handle the consonants. Great, they can both be subclasses of the *Letter* class, and the user will never know the difference, right? Wrong. How is the client supposed to get at these new classes? They've got code that works perfectly well for them by calling *new Letter('a')* and *new Letter('Z')*. Now you're going to make them go through all their code and change these to *new Vowel('a')* and *new Consonant('Z')*? They probably won't be too happy with that. If only you could get new instances of both classes from one method! Well you can, just use a static method in the *Letter* class, it'll do the same one-time checking as the constructor did, and will return an appropriate instance of the right class. And what do you know, it's a factory method! But that still doesn't do your client much good, they still need to go through and change all the *new Letter()*s into *Letter.getLetter()*. Well, sad to say, it's too late to help them at all, unless you just give up your new implementation. But that illustrates the reason for using the factory method right off the bat. One of the key components of good object oriented programming is that you never know exactly where your code will go in the future. By making good use of the abstraction barrier and using encapsulation-friendly programming patterns, such as the factory pattern, you can better prepare yourself—and your client—for future changes to the specific implementation. In particular, it allows you to use a "big hammer" kind of approach to get something done in a perhaps-less-than-ideal but rapid manner in order to meet deadlines or move ahead with testing,. You can then go back later and refine the implementation—the data representation and algorithms—to be faster, smaller, or what-have-you, and as long as you maintained the abstraction barrier between implementor and client and properly encapsulated your implementation, then you can change it without requiring the client to change any of their code. Well now that I'm sure you're a raving advocate for the factory method, let's take a look at how we would implement it for our *Letter* type:

```
public abstract class Letter {

// Factory Method

 public static Letter getLetter(char aTheLetter) {
 // Like before, we do a one time check to see what kind of
 // letter we are dealing with. Only this time, instead of setting
 // a property to track it, we actually have a different class for each
 // of the two letter types.
 if (
 aTheLetter == 'a' ||
 aTheLetter == 'e' ||
 aTheLetter == 'i' ||
 aTheLetter == 'o' ||
 aTheLetter == 'u' ||
 aTheLetter == 'A' ||
 aTheLetter == 'E' ||
 aTheLetter == 'I' ||
 aTheLetter == 'O' ||
 aTheLetter == 'U'
 ) {
   return new Vowel(aTheLetter);
 } else {
   return new Consonant(aTheLetter);
 }
 }


// User facing interface
// We make these methods abstract, thereby requiring all subclasses
// (actually, just all concrete subclasses, that is, non-abstract)
// to implement the methods.

 public abstract boolean isVowel();

 public abstract boolean isConsonant();

 public abstract char getChar();


 // Now we define the two concrete classes for this type,
 // the ones that actually implement the type.

 private static class Vowel extends Letter {
 private char iTheLetter;

 // Constructor
 Vowel(char aTheLetter) {
 this.iTheLetter = aTheLetter;
 }

 // Nice easy implementation of this method!
 public boolean isVowel() {
 return true;
 }

 // This one, too!
 public boolean isConsonant() {
 return false;
 }

 public char getLetter(){
   return iTheLetter;
 }

 } // End local class Vowel
```

```
private static class Consonant extends Letter {
private char iTheLetter;

// Constructor
Consonant(char aTheLetter) {
  this.iTheLetter = aTheLetter;
}

public boolean isVowel() {
  return false;
}

public boolean isConsonant(){
  return true;
}

public char getLetter(){
  return iTheLetter;
}

} // End local class Consonant

} // End toplevel class Letter
```

Several things to note here.

- First, you'll notice the top level class *Letter* is abstract. This is fine because you'll notice that it doesn't actually do anything except define the interface and provide a top level container for the two other classes. However, it's also *important* (not just OK) to make this abstract because we don't want people trying to instantiate the *Letter* class directly. Of course we could solve this problem by making a private constructor, but making the class abstract instead is cleaner, and makes it more obvious that the *Letter* class is not **meant** to be instantiated. It also, as mentioned, allows us to define the user facing interface that the work horse classes need to implement.
- The two nested classes we created are called **local** classes, which is basically the same as an **inner** class except that local classes are static, and inner classes are not. They have to be static so that our static factory method can create them. If they were non static (i.e., dynamic) then they could only be accessed through an instance of the *Letter* class, which we can never have because *Letter* is abstract. Also note that (in Java, anyway) the fields for inner and local classes typically use the "i" (for inner) prefix, as opposed to the "f" (for field) prefix used by top level classes. This is simply a naming convention used by many Java programmers and doesn't actually effect the program.
- The two nested classes that implement the *Letter* data type do not actually have to be local/inner. They could just have easily been top level classes that extend the abstract *Letter* class. However, this is contrary to the point of the factory pattern, which is encapsulation. Top level classes can't be private in Java, because that doesn't make any sense (what are they private to?) and the whole point is that no client has to (or should, really) know how the type is implemented. Making these classes top level allows clients to potentially stumble across them, and worse yet, instantiate them, by-passing the factory pattern all together.
- Lastly, this is not very good code. There's a lot of ways we can make it better to really illustrate the power of the factory pattern. I'll discuss these refactorings briefly, and then show another, more polished, version of the above code which includes a lot of them.

4.3.1 Refactoring the Factory Pattern

Notice that both of the local classes do the same thing in a few places. This is redundant code which is not only more work to write, but it's also highly discouraged in object oriented programming (partially **because** it takes more work to write, but mostly because it's harder to maintain and prone to errors, e.g., you find a bug in the code and change it in one spot, but forget to in another.) Below is a list of redundancies in the above code:

- The field *iTheLetter*
- The method *getLetter()*
- The constructor of each inner class does the same thing.

In addition, as we discovered above, the isVowel() and isConsonant() just happen to always return the opposite of each other for a given instance. However, since this is something of a peculiarity for this particular example, we won't worry about it. The lesson you would learn from us doing that will already be covered in the refactoring of the *getLetter()* method. OK, so we have redundant code in two classes. If you're familiar with abstracting processes, then this is probably a familiar scenario to you. Often, having redundant code in two different classes makes them prime candidates for abstraction, meaning that a new abstract class is created to implement the redundant code, and the two classes simply extend this new abstract class instead of implementing the redundant code. Well what do you know? We already have an abstract super class that our redundant classes have in common. All we have to do is make the super class implement the redundant code, and the other classes will automatically inherit this implementation, as long as we don't override it. So that works fine for the *getLetter()* method, we can move both the method and the *iTheLetter* field up to the abstract parent class. But what about the constructors? Well our constructor takes an argument, so we won't automatically inherit it, that's just the way java works. But we can use the **super** keyword to automatically delegate to the super classes constructor. In other words, we'll implement the constructor in the super class, since that's where the field is anyway, and the other two classes will delegate to this method in their own constructors. For our example, this doesn't save much work, we're replacing a one line assignment with a one line call to **super()**, but in theory, there could be hundred of lines of code in the constructors, and moving it up could be a great help. At this point, you might be a little worried about putting a constructor in the *Letter* class. Didn't I already say not to do that? I thought we didn't want people trying to instantiate *Letter* directly? Don't worry, the class is still abstract. Even if there's a concrete constructor, Java won't let you instantiate an abstract class, because it's abstract, it could have method that are accessible but undefined, and it wouldn't know what to do if such a method was invoked. So putting the constructor in is fine. After making the above refactorings, our code now looks like this:

```
public abstract class Letter {

// Factory Method

 public static Letter getLetter(char aTheLetter){
 if (
 aTheLetter == 'a' ||
 aTheLetter == 'e' ||
 aTheLetter == 'i' ||
 aTheLetter == 'o' ||
 aTheLetter == 'u' ||
```

```
aTheLetter == 'A' ||
aTheLetter == 'E' ||
aTheLetter == 'I' ||
aTheLetter == 'O' ||
aTheLetter == 'U'
) {
  return new Vowel(aTheLetter);
} else {
  return new Consonant(aTheLetter);
}
}


// Our new abstracted field. We'll make it protected so that subclasses can see
it,
// and we rename it from "i" to "f", following our naming convention.
protected char fTheLetter;


// Our new constructor. It can't actually be used to instantiate an instance
// of Letter, but our sub classes can invoke it with super
protected Letter(char aTheLetter) {
this.fTheLetter = aTheLetter;
}

// The new method we're abstracting up to remove redundant code in the sub
classes
public char getChar() {
return this.fTheLetter;
}

// Same old abstract methods that define part of our client facing interface
public abstract boolean isVowel();

public abstract boolean isConsonant();



// The local subclasses with the redundant code moved up.

private static class Vowel extends Letter {

// Constructor delegates to the super constructor
Vowel(char aTheLetter) {
  super(aTheLetter);
}

// Still need to implement the abstract methods
public boolean isVowel()  {
  return true;
}

public boolean isConsonant(){
  return false;
}
} // End local class Vowel


private static class Consonant extends Letter {

Consonant(char aTheLetter){
  super(aTheLetter);
}

public boolean isVowel(){
  return false;
}
```

```
 public boolean isConsonant(){
   return true;
 }
 } // End local class Consonant


} // End toplevel class Letter
```

Note that we made our abstracted field protected. This isn't strictly necessary in this case, we could have left it private, because the subclasses don't actually need to access it at all. In general, *I* prefer to make things protected instead of private, since, as I mentioned, you can never really be sure where a project will go in the future, and you may not want to restrict future implementors (including yourself) unnecessarily. However, many people prefer to default to private and only use protected when they know it's necessary. A major reason for this is the rather peculiar and somewhat unexpected meaning of **protected** in Java, which allows not only subclasses, but *anything* in the same package to access it. This is a bit of a digression, but I think it's a fairly important debate that a good Java programmer should be aware of.

4.4 The Factory Pattern and Parametric Polymorphism

The version of the Java Virtual Machine 5.0 has introduced something called Parametric Polymorphism, which goes by many other names in other languages, including "generic typing" in C++. In order to really understand the rest of this section, you should read that section first. But basically, this means that you can introduce additional parameters into a class--parameters which are set at instantiation--that define the types of certain elements in the class, for instance fields or method return values. This is a very powerful tool which allows programmers to avoid a lot of those nasty **instanceof**s and narrowing castes. However, the implementation of this device in the JVM does not promote the use of the Factory pattern, and in the two do not play well together. This is because Java does not allow methods to be parameterized the way types are, so you cannot dynamically parameterize an instance through a method, only through use of the **new** operator. As an example, imagine a type *Foo* which is parameterized with a single type which we'll call *T*. In java we would write this class like this:

```
class FooT {

} // End class Foo
```

Now we can have instances of *Foo* parameterized by all sorts of types, for instance:

```
FooString fooOverString = new FooString();
FooInteger fooOverInteger = new FooInteger();
```

But let's say we want to use the factory pattern for *Foo*. How do we do that? You could create a different factory method for each type you want to parameterize over, for instance:

```
class FooT {
```

```
  static FooString getFooOverString() {
    return new FooString();
  }

  static FooInteger getFooOverInteger() {
    return new FooInteger();
  }

} // End class Foo
```

But what about something like the *ArrayList* class (in the java.util package)? In the java standard libraries released with 5.0, ArrayList is parameterized to define the type of the object stored in it. We certainly don't want to restrict what kinds of types it can be parameterized with by having to write a factory method for each type. This is often the case with parameterized types: you don't know what types users will want to parameterize with, and you don't want to restrict them, so the factory pattern won't work for that. You are allowed to instantiate a parameterized type in a generic form, meaning you don't specify the parameter at all, you just instantiate it the way you would have before 5.0. But that forces you to give up the parameterization. This is how you do it with generics:

```
class FooT {

  public static E FooE getFoo() {
    return new FooE();
  }

} // End class Foo
```

Implementation in Haskell

```
class Pizza a where
  price :: a - Float

data HamMushroom = HamMushroom
data Deluxe      = Deluxe
data Hawaiian    = Hawaiian

instance Pizza HamMushroom where
  price _ = 8.50

instance Pizza Deluxe where
  price _ = 10.50

instance Pizza Hawaiian where
  price _ = 11.50
```

Usage example:

```
main = print (price Hawaiian)
```

Implementation in ABAP

```
REPORT zz_pizza_factory_test NO STANDARD PAGE HEADING .

TYPES ty_pizza_type TYPE i .

*---------------------------------------------------------------------*
```

```
*       CLASS lcl_pizza DEFINITION
*---------------------------------------------------------------------*
CLASS lcl_pizza DEFINITION ABSTRACT .

  PUBLIC SECTION .

    DATA p_pizza_name TYPE string .

    METHODS get_price ABSTRACT
                      RETURNING value(y_price) TYPE i .

ENDCLASS .                    "lcl_pizza DEFINITION

*---------------------------------------------------------------------*
*       CLASS lcl_ham_and_mushroom_pizza DEFINITION
*---------------------------------------------------------------------*
CLASS lcl_ham_and_mushroom_pizza DEFINITION INHERITING FROM lcl_pizza .

  PUBLIC SECTION .

    METHODS constructor .
    METHODS get_price REDEFINITION .

ENDCLASS .                    "lcl_ham_and_mushroom_pizza DEFINITION

*---------------------------------------------------------------------*
*       CLASS lcl_deluxe_pizza DEFINITION
*---------------------------------------------------------------------*
CLASS lcl_deluxe_pizza DEFINITION INHERITING FROM lcl_pizza .

  PUBLIC SECTION .

    METHODS constructor .
    METHODS get_price REDEFINITION .

ENDCLASS .                    "lcl_ham_and_mushroom_pizza DEFINITION

*---------------------------------------------------------------------*
*       CLASS lcl_hawaiian_pizza DEFINITION
*---------------------------------------------------------------------*
CLASS lcl_hawaiian_pizza DEFINITION INHERITING FROM lcl_pizza .

  PUBLIC SECTION .

    METHODS constructor .
    METHODS get_price REDEFINITION .

ENDCLASS .                    "lcl_ham_and_mushroom_pizza DEFINITION

*---------------------------------------------------------------------*
*       CLASS lcl_pizza_factory DEFINITION
*---------------------------------------------------------------------*
CLASS lcl_pizza_factory DEFINITION .
  PUBLIC SECTION .

    CONSTANTS: BEGIN OF co_pizza_type ,
                 ham_mushroom  TYPE ty_pizza_type VALUE 1 ,
                 deluxe        TYPE ty_pizza_type VALUE 2 ,
                 hawaiian      TYPE ty_pizza_type VALUE 3 ,
               END OF co_pizza_type .


    CLASS-METHODS create_pizza IMPORTING  x_pizza_type TYPE ty_pizza_type
                               RETURNING value(yo_pizza) TYPE REF TO lcl_pizza
                               EXCEPTIONS ex_invalid_pizza_type .


ENDCLASS .                    "lcl_pizza_factory DEFINITION
```

```
*---------------------------------------------------------------------*
*       CLASS lcl_ham_and_mushroom_pizza
*---------------------------------------------------------------------*
CLASS lcl_ham_and_mushroom_pizza IMPLEMENTATION .

  METHOD constructor .
    super-constructor( ) .
    p_pizza_name = 'Ham  Mushroom Pizza'(001) .
  ENDMETHOD .                    "constructor

  METHOD get_price .
    y_price = 850 .
  ENDMETHOD .                    "get_price

ENDCLASS .                    "lcl_ham_and_mushroom_pizza IMPLEMENTATION

*---------------------------------------------------------------------*
*       CLASS lcl_deluxe_pizza IMPLEMENTATION
*---------------------------------------------------------------------*
CLASS lcl_deluxe_pizza IMPLEMENTATION .

  METHOD constructor .
    super-constructor( ) .
    p_pizza_name = 'Deluxe Pizza'(002) .
  ENDMETHOD .                    "constructor

  METHOD get_price .
    y_price = 1050 .
  ENDMETHOD .                    "get_price

ENDCLASS .                    "lcl_deluxe_pizza IMPLEMENTATION

*---------------------------------------------------------------------*
*       CLASS lcl_hawaiian_pizza IMPLEMENTATION
*---------------------------------------------------------------------*
CLASS lcl_hawaiian_pizza IMPLEMENTATION .

  METHOD constructor .
    super-constructor( ) .
    p_pizza_name = 'Hawaiian Pizza'(003) .
  ENDMETHOD .                    "constructor

  METHOD get_price .
    y_price = 1150 .
  ENDMETHOD .                    "get_price

ENDCLASS .                    "lcl_hawaiian_pizza IMPLEMENTATION

*---------------------------------------------------------------------*
*       CLASS lcl_pizza_factory IMPLEMENTATION
*---------------------------------------------------------------------*
CLASS lcl_pizza_factory IMPLEMENTATION .

  METHOD create_pizza .

    CASE x_pizza_type .
      WHEN co_pizza_type-ham_mushroom .
        CREATE OBJECT yo_pizza TYPE lcl_ham_and_mushroom_pizza .
      WHEN co_pizza_type-deluxe .
        CREATE OBJECT yo_pizza TYPE lcl_deluxe_pizza .
      WHEN co_pizza_type-hawaiian .
        CREATE OBJECT yo_pizza TYPE lcl_hawaiian_pizza .
    ENDCASE .

  ENDMETHOD .                    "create_pizza

ENDCLASS .                    "lcl_pizza_factory IMPLEMENTATION
```

```
START-OF-SELECTION .

  DATA go_pizza TYPE REF TO lcl_pizza .
  DATA lv_price TYPE i .

  DO 3 TIMES .

    go_pizza = lcl_pizza_factory=create_pizza( sy-index ) .
    lv_price = go_pizza-get_price( ) .
    WRITE:/ 'Price of', go_pizza-p_pizza_name, 'is £', lv_price LEFT-JUSTIFIED .

  ENDDO .

*Output:
*Price of Ham  Mushroom Pizza is £ 850
*Price of Deluxe Pizza is £ 1.050
*Price of Hawaiian Pizza is £ 1.150
```

Implementation in ActionScript 3.0

```
public class Pizza
{
protected var _price:Number;
public function get price():Number
        {
return _price;
}
}

public class HamAndMushroomPizza extends Pizza
{
public function HamAndMushroomPizza()
        {
_price = 8.5;
}
}

public class DeluxePizza extends Pizza
{
public function DeluxePizza()
        {
_price = 10.5;
}
}

public class HawaiianPizza extends Pizza
{
public function HawaiianPizza()
        {
_price = 11.5;
}
}

public class PizzaFactory
{
static public function createPizza(type:String):Pizza
        {
switch (type)
                {
case "HamAndMushroomPizza":
return new HamAndMushroomPizza();
break;

case "DeluxePizza":
```

```
return new DeluxePizza();
break;

case "HawaiianPizza":
return new HawaiianPizza();
break;

default:
throw new ArgumentError("The pizza type " + type + " is not recognized.");
}
}
}

public class Main extends Sprite
{
public function Main()
        {
for each (var pizza:String in ["HamAndMushroomPizza", "DeluxePizza",
 "HawaiianPizza"])
                {
trace("Price of " + pizza + " is " + PizzaFactory.createPizza(pizza).price);
}
}
}

Output:
Price of HamAndMushroomPizza is 8.5
Price of DeluxePizza is 10.5
Price of HawaiianPizza is 11.5
```

Implementation in VB.NET

```
Imports System

Namespace FactoryMethodPattern
  Public Class Program

    Shared Sub Main()
      OutputPizzaFactory(New LousPizzaStore())
      OutputPizzaFactory(New TonysPizzaStore())

      Console.ReadKey()
    End Sub

    Private Shared Sub OutputPizzaFactory(ByVal factory As IPizzaFactory)
      Console.WriteLine("Welcome to {0}", factory.Name)
      For Each p As Pizza In factory.CreatePizzas
        Console.WriteLine("  {0} - ${1} - {2}", p.GetType().Name, p.Price,
 p.Toppings)
      Next
    End Sub
  End Class

  Public MustInherit Class Pizza
    Protected _toppings As String
    Protected _price As Decimal
    Public ReadOnly Property Toppings() As String
      Get
        Return _toppings
      End Get
    End Property
    Public ReadOnly Property Price() As Decimal
      Get
        Return _price
      End Get
    End Property
```

```
  Public Sub New(ByVal __price As Decimal)
    _price = __price
  End Sub
End Class

Public Interface IPizzaFactory
  ReadOnly Property Name() As String
  Function CreatePizzas() As Pizza()
End Interface

Public Class Pepperoni
  Inherits Pizza
  Public Sub New(ByVal price As Decimal)
    MyBase.New(price)
    _toppings = "Cheese, Pepperoni"
  End Sub
End Class

Public Class Cheese
  Inherits Pizza
  Public Sub New(ByVal price As Decimal)
    MyBase.New(price)
    _toppings = "Cheese"
  End Sub
End Class

Public Class LousSpecial
  Inherits Pizza
  Public Sub New(ByVal price As Decimal)
    MyBase.New(price)
    _toppings = "Cheese, Pepperoni, Ham, Lou's Special Sauce"
  End Sub
End Class

Public Class TonysSpecial
  Inherits Pizza
  Public Sub New(ByVal price As Decimal)
    MyBase.New(price)
    _toppings = "Cheese, Bacon, Tomatoes, Tony's Special Sauce"
  End Sub
End Class

Public Class LousPizzaStore
  Implements IPizzaFactory

  Public Function CreatePizzas() As Pizza() Implements
IPizzaFactory.CreatePizzas
    Return New Pizza() {New Pepperoni(6.99D), New Cheese(5.99D), New
LousSpecial(7.99D)}
  End Function

  Public ReadOnly Property Name() As String Implements IPizzaFactory.Name
    Get
      Return "Lou's Pizza Store"
    End Get
  End Property
End Class

Public Class TonysPizzaStore
  Implements IPizzaFactory

  Public Function CreatePizzas() As Pizza() Implements
IPizzaFactory.CreatePizzas
    Return New Pizza() {New Pepperoni(6.5D), New Cheese(5.5D), New
TonysSpecial(7.5D)}
  End Function
```

```
    Public ReadOnly Property Name() As String Implements IPizzaFactory.Name
      Get
        Return "Tony's Pizza Store"
      End Get
    End Property
  End Class

End Namespace

Output:
Welcome to Lou's Pizza Store
  Pepperoni - $6.99 - Cheese, Pepperoni
  Cheese - $5.99 - Cheese
  LousSpecial - $7.99 - Cheese, Pepperoni, Ham, Lou's Special Sauce
Welcome to Tony's Pizza Store
  Pepperoni - $6.5 - Cheese, Pepperoni
  Cheese - $5.5 - Cheese
  TonysSpecial - $7.5 - Cheese, Bacon, Tomatoes, Tony's Special Sauce
```

Implementation in Common Lisp

In Common Lisp[2], factory methods are not really needed, because classes and class names are first class values.

```
(defclass pizza ()
  ((price :accessor price)))

(defclass ham-and-mushroom-pizza (pizza)
  ((price :initform 850)))

(defclass deluxe-pizza (pizza)
  ((price :initform 1050)))

(defclass hawaiian-pizza (pizza)
  ((price :initform 1150)))

(defparameter *pizza-types*
  (list 'ham-and-mushroom-pizza
        'deluxe-pizza
        'hawaiian-pizza))

(loop for pizza-type in *pizza-types*
      do (format t "~%Price of ~a is ~a"
                 pizza-type
                 (price (make-instance pizza-type))))

Output:
Price of HAM-AND-MUSHROOM-PIZZA is 850
Price of DELUXE-PIZZA is 1050
Price of HAWAIIAN-PIZZA is 1150
```

Implementation in Java

```
abstract class Pizza {
    public abstract int getPrice(); // count the cents
}

class HamAndMushroomPizza extends Pizza {
    public int getPrice() {
        return 850;
```

2 http://en.wikibooks.org/wiki/Common_Lisp

```
    }
}

class DeluxePizza extends Pizza {
    public int getPrice() {
        return 1050;
    }
}

class HawaiianPizza extends Pizza {
    public int getPrice() {
        return 1150;
    }
}

class PizzaFactory {
    public enum PizzaType {
        HamMushroom,
        Deluxe,
        Hawaiian
    }

    public static Pizza createPizza(PizzaType pizzaType) {
        switch (pizzaType) {
            case HamMushroom:
                return new HamAndMushroomPizza();
            case Deluxe:
                return new DeluxePizza();
            case Hawaiian:
                return new HawaiianPizza();
        }
        throw new IllegalArgumentException("The pizza type " + pizzaType + " is
 not recognized.");
    }
}

class PizzaLover {
    /*
     * Create all available pizzas and print their prices
     */
    public static void main (String args[]) {
        for (PizzaFactory.PizzaType pizzaType : PizzaFactory.PizzaType.values())
 {
            System.out.println("Price of " + pizzaType + " is " +
 PizzaFactory.createPizza(pizzaType).getPrice());
        }
    }
}

Output:
Price of HamMushroom is 850
Price of Deluxe is 1050
Price of Hawaiian is 1150
```

Implementation in Javascript

This example in JavaScript[3] uses Firebug[4] console to output information.

```
/**
 * Extends parent class with child. In Javascript, the keyword "extends" is not
 * currently implemented, so it must be emulated.
```

3 http://en.wikibooks.org/wiki/JavaScript

4 http://en.wikibooks.org/w/index.php?title=Firebug_(web_development)&action=edit&redlink=1

```
 * Also it is not recommended to use keywords for future use, so we name this
 * function "extends" with capital E. Javascript is case-sensitive.
 *
 * @param  function  parent constructor function
 * @param  function  (optional) used to override default child constructor
 function
 */
function Extends(parent, childConstructor) {

  var F = function () {};

  F.prototype = parent.prototype;

  var Child = childConstructor || function () {};
  Child.prototype = new F();
  Child.prototype.constructor = Child;
  Child.parent = parent.prototype;

  // return instance of new object
  return Child;
}

/**
 * Abstract Pizza object constructor
 */
function Pizza() {
  throw new Error('Cannot instatiate abstract object!');
}
Pizza.prototype.price = 0;
Pizza.prototype.getPrice = function () {
  return this.price;
}

var HamAndMushroomPizza = Extends(Pizza);
HamAndMushroomPizza.prototype.price = 8.5;

var DeluxePizza = Extends(Pizza);
DeluxePizza.prototype.price = 10.5;

var HawaiianPizza = Extends(Pizza);
HawaiianPizza.prototype.price = 11.5;


var PizzaFactory = {
  createPizza: function (type) {
    var baseObject = 'Pizza';
    var targetObject = type.charAt(0).toUpperCase() + type.substr(1);

    if (typeof window[targetObject + baseObject] === 'function') {
      return new window[targetObject + baseObject];
    }
    else {
      throw new Error('The pizza type ' + type + ' is not recognized.');
    }
  }
};

//var price = PizzaFactory.createPizza('deluxe').getPrice();
var pizzas = ['HamAndMushroom', 'Deluxe', 'Hawaiian'];
for (var i in pizzas) {
  console.log('Price of ' + pizzas[i] + ' is ' +
 PizzaFactory.createPizza(pizzas[i]).getPrice());
}
```

Output

```
Price of HamAndMushroom is 8.50
Price of Deluxe is 10.50
Price of Hawaiian is 11.50
```

Implementation in Perl

```
package Pizza;
use Moose;
has price = (is = "rw", isa = "Num", builder = "_build_price" );

package HamAndMushroomPizza;
use Moose; extends "Pizza";
sub _build_price { 8.5 }

package DeluxePizza;
use Moose; extends "Pizza";
sub _build_price { 10.5 }

package HawaiianPizza;
use Moose; extends "Pizza";
sub _build_price { 11.5 }

package PizzaFactory;

sub create {
    my ( $self, $type ) = @_;
    return ($type . "Pizza")-new;
}

package main;
for my $type ( qw( HamAndMushroom Deluxe Hawaiian ) ) {
    printf "Price of %s is %.2f\n", $type, PizzaFactory-create( $type )-price;
}
```

Factories are not really needed for this example in Perl, and this may be written more concisely:

```
package Pizza;
use Moose;
has price = (is = "rw", isa = "Num", builder = "_build_price" );

package HamAndMushroomPizza;
use Moose; extends "Pizza";
sub _build_price { 8.5 }

package DeluxePizza;
use Moose; extends "Pizza";
sub _build_price { 10.5 }

package HawaiianPizza;
use Moose; extends "Pizza";
sub _build_price { 11.5 }

package main;
for my $type ( qw( HamAndMushroom Deluxe Hawaiian ) ) {
    printf "Price of %s is %.2f\n", $type, ($type . "Pizza")-new-price;
}
```

Output

```
Price of HamAndMushroom is 8.50
Price of Deluxe is 10.50
Price of Hawaiian is 11.50
```

Implementation in Python

```
#
# Pizza
#
class Pizza(object):
    def __init__(self):
        self._price = None

    def get_price(self):
        return self._price

class HamAndMushroomPizza(Pizza):
    def __init__(self):
        self._price = 8.5

class DeluxePizza(Pizza):
    def __init__(self):
        self._price = 10.5

class HawaiianPizza(Pizza):
    def __init__(self):
        self._price = 11.5

#
# PizzaFactory
#
class PizzaFactory(object):
    @staticmethod
    def create_pizza(pizza_type):
        if pizza_type == 'HamMushroom':
            return HamAndMushroomPizza()
        elif pizza_type == 'Deluxe':
            return DeluxePizza()
        elif pizza_type == 'Hawaiian':
            return HawaiianPizza()

if __name__ == '__main__':
    for pizza_type in ('HamMushroom', 'Deluxe', 'Hawaiian'):
        print 'Price of {0} is {1}'.format(pizza_type,
 PizzaFactory.create_pizza(pizza_type).get_price())
```

As in Perl, Common Lisp and other dynamic languages, factories of the above sort aren't really necessary, since classes are first-class objects and can be passed around directly, leading to this more natural version:

```
#
# Pizza
#
class Pizza(object):
    def __init__(self):
        self._price = None

    def get_price(self):
        return self._price

class HamAndMushroomPizza(Pizza):
```

```
    def __init__(self):
        self._price = 8.5

class DeluxePizza(Pizza):
    def __init__(self):
        self._price = 10.5

class HawaiianPizza(Pizza):
    def __init__(self):
        self._price = 11.5

if __name__ == '__main__':
    for pizza_class in (HamAndMushroomPizza, DeluxePizza, HawaiianPizza):
        print 'Price of {0} is {1}'.format(pizza_class.__name__,
 pizza_class().get_price())
```

Note in the above that the classes themselves are simply used as values, which `pizza_class` iterates over. The class gets created simply by treating it as a function. In this case, if `pizza_class` holds a class, then `pizza_class()` creates a new object of that class. Another way of writing the final clause, which sticks more closely to the original example and uses strings instead of class objects, is as follows:

```
if __name__ == '__main__':
    for pizza_type in ('HamAndMushroom', 'Deluxe', 'Hawaiian'):
        print 'Price of {0} is {1}'.format(pizza_type, eval(pizza_type +
 'Pizza')().get_price())
```

In this case, the correct class name is constructed as a string by adding `'Pizza'`, and `eval` is called to turn it into a class object.

Implementation in PHP

```
?php

abstract class Pizza
{
    protected $_price;
    public function getPrice()
    {
        return $this-_price;
    }
}

class HamAndMushroomPizza extends Pizza
{
    protected $_price = 8.5;
}

class DeluxePizza extends Pizza
{
    protected $_price = 10.5;
}

class HawaiianPizza extends Pizza
{
    protected $_price = 11.5;
}

class PizzaFactory
{
    public static function createPizza($type)
    {
```

```
        $baseClass = 'Pizza';
        $targetClass = ucfirst($type).$baseClass;

        if (class_exists($targetClass)  is_subclass_of($targetClass,
 $baseClass))
            return new $targetClass;
        else
            throw new Exception("The pizza type '$type' is not recognized.");
    }
}

$pizzas = array('HamAndMushroom','Deluxe','Hawaiian');
foreach($pizzas as $p) {
    printf(
        "Price of %s is %01.2f".PHP_EOL ,
        $p ,
        PizzaFactory::createPizza($p)-getPrice()
    );
}


// Output:
// Price of HamAndMushroom is 8.50
// Price of Deluxe is 10.50
// Price of Hawaiian is 11.50

?
```

Implementation in Delphi

```
program FactoryMethod;

{$APPTYPE CONSOLE}

uses
  SysUtils;

type

// Product
TProduct = class(TObject)
public
  function GetName(): string; virtual; abstract;
end;

// ConcreteProductA
TConcreteProductA = class(TProduct)
public
  function GetName(): string; override;
end;

// ConcreteProductB
TConcreteProductB = class(TProduct)
public
  function GetName(): string; override;
end;

// Creator
TCreator = class(TObject)
public
  function FactoryMethod(): TProduct; virtual; abstract;
end;

// ConcreteCreatorA
TConcreteCreatorA = class(TCreator)
public
```

```
  function FactoryMethod(): TProduct; override;
end;

// ConcreteCreatorB
TConcreteCreatorB = class(TCreator)
public
  function FactoryMethod(): TProduct; override;
end;

{ ConcreteProductA }
function TConcreteProductA.GetName(): string;
begin
  Result := 'ConcreteProductA';
end;

{ ConcreteProductB }
function TConcreteProductB.GetName(): string;
begin
  Result := 'ConcreteProductB';
end;

{ ConcreteCreatorA }
function TConcreteCreatorA.FactoryMethod(): TProduct;
begin
  Result := TConcreteProductA.Create();
end;

{ ConcreteCreatorB }
function TConcreteCreatorB.FactoryMethod(): TProduct;
begin
  Result := TConcreteProductB.Create();
end;

const
  Count = 2;

var
  Creators: array[1..Count] of TCreator;
  Product: TProduct;
  I: Integer;

begin
  // An array of creators
  Creators[1] := TConcreteCreatorA.Create();
  Creators[2] := TConcreteCreatorB.Create();

  // Iterate over creators and create products
  for I := 1 to Count do
  begin
    Product := Creators[I].FactoryMethod();
    WriteLn(Product.GetName());
    Product.Free();
  end;

  for I := 1 to Count do
    Creators[I].Free();

  ReadLn;
end.
```

5 Prototype

The prototype pattern is used 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-coded 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.

5.1 Structure

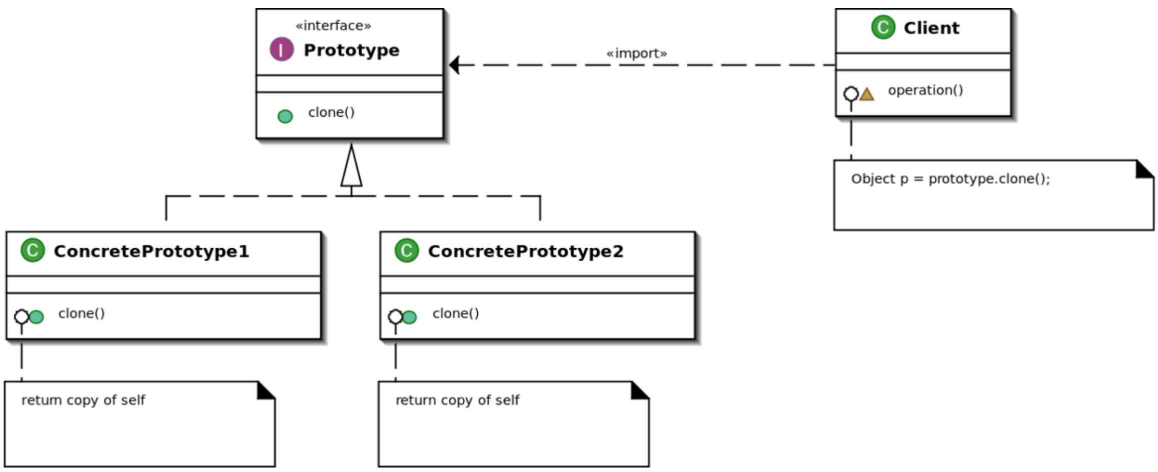


Figure 5 UML class diagram describing the Prototype design pattern

5.2 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* that 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 that 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.

5.3 Implementations

It 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.

Implementation in C#

```
//Note: In this example ICloneable interface (defined in .Net Framework) acts as
 Prototype

class ConcretePrototype : ICloneable
{
    public int X { get; set; }

    public ConcretePrototype(int x)
    {
        this.X = x;
    }

    public void PrintX()
    {
        Console.WriteLine("Value :" + X);
    }

    public object Clone()
    {
        return this.MemberwiseClone();
    }
```

```
}

/**
 * Client code
 */
public class PrototypeTest
{
    public static void Main()
    {
        var prototype = new ConcretePrototype(1000);

        for (int i = 1; i  10; i++)
        {
            ConcretePrototype tempotype = prototype.Clone() as
 ConcretePrototype;

            // Usage of values in prototype to derive a new value.
            tempotype.X *= i;
            tempotype.PrintX();
        }
        Console.ReadKey();
    }
}

/*
 **Code output**

Value :1000
Value :2000
Value :3000
Value :4000
Value :5000
Value :6000
Value :7000
Value :8000
Value :9000
*/
```

Implementation in C++

```cpp
// Prototype
class Prototype
{
public:
    virtual ~Prototype() { }

    virtual Prototype* clone() const = 0;
};

// Concrete prototype
class ConcretePrototype : public Prototype
{
public:
    ConcretePrototype(int x) : x_(x) { }

    ConcretePrototype(const ConcretePrototype p) : x_(p.x_) { }

    virtual ConcretePrototype* clone() const { return new
 ConcretePrototype(*this); }

    void setX(int x) { x_ = x; }

    int getX() const { return x_; }

    void printX() const { std::cout  "Value :"  x_  std::endl; }
```

```
private:
    int x_;
};

// Client code
void prototype_test()
{
    Prototype* prototype = new ConcretePrototype(1000);
    for (int i = 1; i  10; i++) {
        ConcretePrototype* tempotype =
 static_castConcretePrototype*(prototype-clone());
        tempotype-setX(tempotype-getX() * i);
        tempotype-printX();
        delete tempotype;
    }
    delete prototype;
}
/*
 **Code output**

Value :1000
Value :2000
Value :3000
Value :4000
Value :5000
Value :6000
Value :7000
Value :8000
Value :9000
*/
```

Implementation in Java

```
/**
 * Prototype class
 */
interface Prototype {
    void setX(int x);

    void printX();

    int getX();
}

/**
 * Implementation of prototype class
 */
class PrototypeImpl implements Prototype, Cloneable {
    private int x;

    /**
     * Constructor
     */
    public PrototypeImpl(int x) {
        setX(x);
    }

    @Override
    public void setX(int x) {
        this.x = x;
    }

    @Override
    public void printX() {
        System.out.println("Value: " + getX());
    }
```

```
    @Override
    public int getX() {
        return x;
    }

    @Override
    public PrototypeImpl clone() throws CloneNotSupportedException {
        return (PrototypeImpl) super.clone();
    }
}

/**
 * Client code
 */
public class PrototypeTest {
    public static void main(String args[]) throws CloneNotSupportedException {
        PrototypeImpl prototype = new PrototypeImpl(1000);

        for (int y = 1; y  10; y++) {
            // Create a defensive copy of the object to allow safe mutation
            Prototype tempotype = prototype.clone();

            // Derive a new value from the prototype's "x" value
            tempotype.setX(tempotype.getX() * y);
            tempotype.printX();
        }
    }
}

/*
 **Code output**

Value: 1000
Value: 2000
Value: 3000
Value: 4000
Value: 5000
Value: 6000
Value: 7000
Value: 8000
Value: 9000
*/
```

Implementation in PHP

```
?php
class ConcretePrototype {
    protected $x;

    public function __construct($x) {
        $this-x = (int) $x;
    }

    public function printX() {
        echo sprintf('Value: %5d' . PHP_EOL, $this-x);
    }

    public function setX($x) {
        $this-x *= (int) $x;
    }

    public function __clone() {
        /*
         * This method is not required for cloning, although when implemented,
         * PHP will trigger it after the process in order to permit you some
```

```
         * change in the cloned object.
         *
         * Reference: http://php.net/manual/en/language.oop5.cloning.php
         */
        // $this-x = 1;
    }
}

/**
 * Client code
 */
$prototype = new ConcretePrototype(1000);

foreach (range(1, 10) as $i) {
    $tempotype = clone $prototype;
    $tempotype-setX($i);
    $tempotype-printX();
}

/*
 **Code output**

Value:  1000
Value:  2000
Value:  3000
Value:  4000
Value:  5000
Value:  6000
Value:  7000
Value:  8000
Value:  9000
Value: 10000
*/
```

6 Print version

The term ***Singleton*** refers to an object that can be instantiated only once. In programming languages like Java, you have to create an instance of an object type (commonly called a Class) to use it throughout the code. Let's take for example this code:

```
Animal dog;                 // Declaring the object type
dog = new Animal();         // Instantiating an object
```

This can also be written in a single line to save space.

```
Animal dog = new Animal();   // Both declaring and instantiating
```

At times, you need more than one instance of the same object type. You can create multiple instances of the same object type. Take for instance:

```
Animal cat = new Animal();
Animal dog = new Animal();
```

Now that I have two instances of the same object type, I can use both `dog` and `cat` instances separately in my program. Any changes to `dog` would not affect the `cat` instance because both of them have been created in separate memory spaces. To see whether these objects actually are different we do something like this:

```
System.out.println(dog.equals(cat));  // output: false
```

The code returns `false` because both the objects are different. Contrary to this behavior, the **Singleton** behaves differently. A Singleton object type can not be instantiated yet you can obtain an instance of the object type. Let us create a normal object using Java.

```
class NormalObject {
    public NormalObject() {
    }
}
```

What we have done here is that we have created a class (object type) to identify our object. Within the braces of the class is a single method with the same name as that of the class (methods can be identified by the usage of parentheses at the end of their names). Methods that have the same name as that of the class and that do not have a return type are called **Constructors** in OOP syntax. To create an instance of the class, the code can not be much simpler.

```
class TestHarness {
    public static void main(String[] args) {
         NormalObject object = new NormalObject();
    }
}
```

Note that to encourage the instantiation of this object, the constructor is called. The constructor as in this case can be called outside the class parenthesis and into another class definition because it is declared a public accessor while creating the Constructor in the above example. Now we will create the Singleton object. You just have to change one thing to adhere to the Singleton design pattern: Make your Constructor's accessor *private*.

```
class SingletonObject {
    private SingletonObject() {
    }
}
```

Notice the constructor's accessor. This time around it has been declared **private**. Just by changing it to private, you have applied a great change to your object type. Now you can not instantiate the object type. Go ahead try out the following code.

```
class TestHarness {
    public static void main(String[] args) {
         SingletonObject object = new SingletonObject();
    }
}
```

The code returns an error because private class members can not be accessed from outside the class itself and in another class. This way you have disabled the instantiation process for an object type. However, you have to find a way of obtaining an instance of the object type. Let's do some changes here.

```
class SingletonObject {
    private static SingletonObject object;

    private SingletonObject() {
       // Instantiate the object.
    }

    public static SingletonObject getInstance() {
        if (object == null) {
            object = new SingletonObject(); // Create the object for the first
and last time
        }
        return object;
    }
}
```

The changes involve adding a static class member called `object` and a `public static` method that can be accessible outside the scope of the class by using the name of the Class. To see how we can obtain the instance, let's code:

```
class TestHarness {
    public static void main(String[] args) {
         SingletonObject object = SingletonObject.getInstance();
```

```
    }
}
```

This way you control the creation of objects derived from your class. But we have still not unearthed the final and interesting part of the whole process. Try creating multiple instances and see what happens.

```
class TestHarness {
    public static void main(String[] args) {
         SingletonObject object1 = SingletonObject.getInstance();
         SingletonObject object2 = SingletonObject.getInstance();
    }
}
```

Unlike multiple instances of normal object types, multiple instances of a Singleton are all actually the same object instance. To validate the concept in Java, we try:

```
System.out.println(object1.equals(object2)); // output: true
```

The code returns `true` because both object declarations are actually referring to the same object. So, in summarizing the whole concept, a **Singleton** can be defined as an object that can not be instantiated more than once. Typically it is obtained using a static custom implementation. In some applications, it is appropriate to enforce a single instance of an object, for example: window managers, print spoolers, database access, and file systems.

Singleton & multithreading

Java uses multithreading concept, to run/execute any program. Consider the class `SingletonObject` discussed above. Call to the method `getInstance()` by more than one thread at any point of time might create two instances of `SingletonObject`, thus defeating the whole purpose of creating the singleton pattern. To make singleton thread safe, we have different options: 1. Synchronize the method `getInstance()`, which would look like:

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

Synchronizing the method guarantees that a call to the method cannot be interrupted. 2. Synchronize the block, which would look like:

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

Synchronizing the block does the same as the synchronizing on the method. The only advantage is that you can synchronize the block on another class/object. 3. Another approach would be to create a singleton instance as shown below:

```
class SingletonObject {
    public final static SingletonObject object = new SingletonObject();

    private SingletonObject() {
       // Exists only to avoid instantiation.
    }

    public static SingletonObject getInstance() {
        object = SingletonObject.object;
        return object;
    }
}
```

Implementation in Scala

6.1 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. Singletons are declared just like classes except "object" replaces the keyword "class".

```
object MySingleton {
  println("Creating the singleton")

  val i : Int = 0
}
```

Implementation in Java

6.2 Traditional simple way using synchronization

This solution is thread-safe without requiring special language constructs:

```
public class Singleton {
  public volatile static Singleton singleton; // volatile is needed so that
 multiple thread can reconcile the instance

  private Singleton(){}

  public static Singleton getSingleton() { // synchronized keyword has been
 removed from here
    if (singleton == null) { // needed because once there is singleton available
 no need to aquire monitor again  again as it is costly
      synchronized(Singleton.class) {
        if (singleton == null) { // this is needed if two threads are waiting at
 the monitor at the time when singleton was getting instantiated
          singleton = new Singleton();
        }
      }
```

```
    }
    return singleton;
  }

}
```

6.3 Initialization on Demand Holder Idiom

This technique is as lazy as possible, and works in all known versions of Java. It takes advantage of language guarantees about class initialization, and will therefore work correctly in all Java-compliant compilers and virtual machines. The nested class is referenced when getInstance() is called making this solution thread-safe without requiring special language constructs.

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

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

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

6.4 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" 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;
}
```

Implementation in D

Singleton pattern in D programming language

```
import std.stdio;
import std.string;
```

```
class Singleton(T) {
    private static T instance;
    public static T opCall() {
        if(instance is null) {
            instance = new T;
        }
        return instance;
    }
}

class Foo {
    public this() {
        writefln("Foo Constructor");
    }
}

void main(){
    Foo a = Singleton!(Foo)();
    Foo b = Singleton!(Foo)();
}
```

Or in this manner

```
// this class should be in a package to make private this() not visible
class Singleton {
    private static Singleton instance;

    public static Singleton opCall() {
        if(instance is null) {
            instance = new Singleton();
        }
        return instance;
    }

    private this() {
        writefln("Singleton constructor");
    }
}

void main(){
    Singleton a = Singleton();
    Singleton b = Singleton();
}
```

Implementation in PHP 5

Singleton pattern in PHP 5:

```
?php
class Singleton {

  // object instance
  private static $instance;

  // The protected construct prevents instantiating the class externally.  The
 construct can be
  // empty, or it can contain additional instructions...
  // This should also be final to prevent extending objects from overriding the
 constructor with
  // public.
  protected final function __construct() {
    ...
  }
```

```
  // The clone and wakeup methods prevents external instantiation of copies of
 the Singleton class,
  // thus eliminating the possibility of duplicate objects.  The methods can be
 empty, or
  // can contain additional code (most probably generating error messages in
 response
  // to attempts to call).
  public function __clone() {
    trigger_error('Clone is not allowed.', E_USER_ERROR);
  }

  public function __wakeup() {
    trigger_error('Deserializing is not allowed.', E_USER_ERROR);
  }

  // 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();

?
```

Implementation in 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.

```
package {
public class Singleton  {

private static var _instance:Singleton = new Singleton();

public function Singleton () {
           if (_instance){
       throw new Error(
                         "Singleton can only be accessed through
 Singleton.getInstance()"
                     );
                    }
}

public static function getInstance():Singleton {
return _instance;
}
```

```
}
}
```

Implementation in 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.

Implementation in 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. Note the explicit static constructor which disables beforefieldinit. See [1]

```
/// Class implements singleton pattern.
```

1 `http://www.yoda.arachsys.com/csharp/beforefieldinit.html`

```
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();

          // Explicit static constructor to disable beforefieldinit
          static SingletonCreator() { }

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

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

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. More about generic singleton solution in C#: [2]

```
/// Parent for singleton

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

    protected Singleton() { }

    private sealed class SingletonCreatorS where S : class, new()
    {
      private static readonly S instance = new S();
```

2 http://www.c-sharpcorner.com/UploadFile/snorrebaard/GenericSingleton11172008110419AM/GenericSingleton.aspx

```
      //explicit static constructor to disable beforefieldinit
      static SingletonCreator() { }

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

    public static T Instance
    {
      get { return SingletonCreatorT.CreatorInstance; }
    }

  }


/// Concrete Singleton

public class TestClass : SingletonTestClass
{
    public string TestProc()
    {
        return "Hello World";
    }
}

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

Implementation in Delphi

As described by James Heyworth in a paper presented to the Canberra PC Users Group Delphi SIG on 11/11/1996, there are several examples of the Singleton pattern built into the Delphi Visual Component Library. This unit demonstrates the techniques that were used in order to create both a Singleton component and a Singleton object:

```
unit Singletn;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type

  TCSingleton = class(TComponent)
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

  TOSingleton = class(TObject)
  public
    constructor Create;
    destructor Destroy; override;
  end;

var
  Global_CSingleton: TCSingleton;
```

```
  Global_OSingleton: TOSingleton;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Design Patterns', [TCSingleton]);
end;

{ TCSingleton }

constructor TCSingleton.Create(AOwner: TComponent);
begin
  if Global_CSingleton  nil then
    {NB could show a message or raise a different exception here}
    Abort
  else begin
    inherited Create(AOwner);
    Global_CSingleton := Self;
  end;
end;

destructor TCSingleton.Destroy;
begin
  if Global_CSingleton = Self then
    Global_CSingleton := nil;
  inherited Destroy;
end;


{ TOSingleton }

constructor TOSingleton.Create;
begin
if Global_OSingleton  nil then
    {NB could show a message or raise a different exception here}
    Abort
  else
    Global_OSingleton := Self;
end;

destructor TOSingleton.Destroy;
begin
  if Global_OSingleton = Self then
    Global_OSingleton := nil;
  inherited Destroy;
end;

procedure FreeGlobalObjects; far;
begin
  if Global_CSingleton  nil then
    Global_CSingleton.Free;
  if Global_OSingleton  nil then
    Global_OSingleton.Free;
end;

begin
  AddExitProc(FreeGlobalObjects);
end.
```

Implementation in Python

The desired properties of the Singleton pattern can most simply be encapsulated in Python by defining a module, containing module-level variables and functions. To use this modular Singleton, client code merely imports the module to access its attributes and functions in the normal manner. This sidesteps many of the wrinkles in the explicitly-coded versions below, and has the singular advantage of requiring zero lines of code to implement. 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.* 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):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, 'self'):
            cls.self = object.__new__(cls)
        return cls.self

#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. A more elegant approach using metaclasses was also suggested.

```
class SingletonType(type):
    def __call__(cls):
        if getattr(cls, '__instance__', None) is None:
            instance = cls.__new__(cls)
            instance.__init__()
            cls.__instance__ = instance
        return cls.__instance__

# Usage
class Singleton(object):
    __metaclass__ = SingletonType

    def __init__(self):
        print '__init__:', self

class OtherSingleton(object):
    __metaclass__ = SingletonType

    def __init__(self):
        print 'OtherSingleton __init__:', self

# Tests
s1 = Singleton()
s2 = Singleton()
assert s1
assert s2
assert s1 is s2

os1 = OtherSingleton()
os2 = OtherSingleton()
assert os1
assert os2
assert os1 is os2
```

Implementation in Perl

In Perl version 5.10 or newer a state variable can be used.

```
package MySingletonClass;
use strict;
use warnings;
use 5.010;

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

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

In older Perls, just use a global variable.

```
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[3] extension module.

Implementation in Ruby

In Ruby, just include the Singleton module from the standard library into the class.

```
require 'singleton'

class Example
  include Singleton
end

# Access to the instance:
Example.instance
```

Implementation in 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.
```

3 http://search.cpan.org/perldoc?MooseX::Singleton

```
class lcl_Singleton implementation.

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

endclass.
```

7 Adapter

The adapter pattern is used when a client class has to call an incompatible provider class. Let's imagine a MailingClient class that needs to call a method on the LegacyEmployee class:

MailingClient
+firstName: String
+lastName: String

IEmployee
+mailEmployee(firstName: String, lastName: String)

LegacyEmployee
+mailEmployee(fullName: String)

`MailingClient` already calls classes that implement the `IEmployee` interface but the `LegacyEmployee` doesn't implement it. We could add a new method to `LegacyEmployee` to implement the `IEmployee` interface but `LegacyEmployee` is legacy code and can't be changed. We could modify the `MailingClient` class to call `LegacyEmployee` but it needs to change every call. The formatting code would be duplicated everywhere. Moreover, `MailingClient` won't be able to call other provider class that implement the `IEmployee` interface any more. So the solution is to code the formatting code in another independent class, an adapter, also called a *wrapper class*:

IEmployee
+mailEmployee(firstName: String, lastName: String)

MailingClient
+firstName: String
+lastName: String

→

EmployeeAdapter
+mailEmployee(firstName: String, lastName: String)

→

LegacyEmployee
+mailEmployee(fullName: String)

`EmployeeAdapter` implements the `IEmployee` interface. `MailingClient` calls `EmployeeAdapter`. `EmployeeAdapter` formats the data and calls `LegacyEmployee`. This type of adapter is called an *object adapter*. The other type of adapter is the *class adapter*.

To do:
Describe the class adapter.

7.1 Examples

The WebGL-2D[1] is a JavaScript library that implements the adapter pattern. This library is used for the HTML5 canvas element. The canvas element has two interfaces: 2d and WebGL. The first one is very simple to use and the second is much more complex but optimized and faster. The WebGL-2D 'adapts' the WebGL interface to the 2d interface, so that the client calls the 2d interface only.

7.2 Cost

Think twice before implementing this pattern. This pattern should not be planned at design time. If you plan to use it for a project from scratch, this means that you don't understand this pattern. It should be used only with legacy code. It is the least bad solution.

7.2.1 Creation

Its implementation is easy but can be expensive. You should not have to refactor the code as the client and the provider should not be able to work together yet.

7.2.2 Maintenance

This is the worst part. Most of the code has redundancies (but less than without the pattern). The modern interface should always provide as much information as the legacy interface needs to work. If one information is missing on the modern interface, it can call the pattern into question.

7.2.3 Removal

This pattern can be easily removed as automatic refactoring operations can easily remove its existence.

7.3 Advices

- Put the *adapter* term in the name of the adapter class to indicate the use of the pattern to the other developers.

1 `https://github.com/corbanbrook/webgl-2d`

7.4 Implementation

7.4.1 Object Adapter

Implementation in Java

Our company has been created by a merger. One list of employees is available in a database you can access via the `CompanyAEmployees` class:

```
/**
 * Employees of the Company A.
 */
public class CompanyAEmployees {
  /**
   * Retrieve the employee information from the database.
   *
   * @param sqlQuery The SQL query.
   * @return The employee object.
   */
  public Employee getEmployee(String sqlQuery) {
      Employee employee = null;
```

```
      // Execute the request.

      return employee;
    }
}
```

One list of employees is available in a LDAP you can access via the `CompanyBEmployees` class:

```
/**
 * Employees of the Company B.
 */
public class CompanyBEmployees {
  /**
   * Retrieve the employee information from the LDAP.
   *
   * @param sqlQuery The SQL query.
   * @return The employee object.
   */
```

```
  public Employee getEmployee(String distinguishedName) {
      Employee employee = null;

      // Call the LDAP.

      return employee;
    }
}
```

To access both to the former employees of the company A and the former employees of the company B, we define an interface that will be used by two adapters, `EmployeeBrowser`:

```
/**
 * Retrieve information about the employees.
 */
interface EmployeeBrowser {
  /**
   * Retrieve the employee information.
   *
```

```
   * @param direction The employee direction.

   * @param division The employee division.

   * @param departement The employee departement.

   * @param service The employee service.

   * @param firstName The employee firstName.

   * @param lastName The employee lastName.

   *

   * @return The employee object.

   */

  Employee getEmployee(String direction, String division, String department,
 String service, String firstName, String lastName);

}
```

We create an adapter for the code of the former company A, `CompanyAAdatper`:

```
/**

 * Adapter for the company A legacy code.

 */

public class CompanyAAdatper implements EmployeeBrowser {
```

```
/**
 * Retrieve the employee information.
 *
 * @param direction The employee direction.
 * @param division The employee division.
 * @param department The employee department.
 * @param service The employee service.
 * @param firstName The employee firstName.
 * @param lastName The employee lastName.
 *
 * @return The employee object.
 */
public Employee getEmployee(String direction, String division, String
department, String service, String firstName, String lastName) {
  String distinguishedName = "SELECT *"
    + " FROM t_employee as employee"
    + " WHERE employee.company= 'COMPANY A'"
    + " AND employee.direction = " + direction
```

```
      + " AND employee.division = " + division
      + " AND employee.department = " + department
      + " AND employee.service = " + service
      + " AND employee.firstName = " + firstName
      + " AND employee.lastName = " + lastName;

    CompanyAEmployees companyAEmployees = new CompanyAEmployees();
    return companyAEmployees.getEmployee(distinguishedName);
  }
}
```

We create an adapter for the code of the former company B, `CompanyBAdatper`:

```
/**
 * Adapter for the company B legacy code.
 */
public class CompanyBAdatper implements EmployeeBrowser {
  /**
```

```
  * Retrieve the employee information.

  *

  * @param direction The employee direction.

  * @param division The employee division.

  * @param department The employee department.

  * @param service The employee service.

  * @param firstName The employee firstName.

  * @param lastName The employee lastName.

  *

  * @return The employee object.

  */

 public Employee getEmployee(String direction, String division, String
department, String service, String firstName, String lastName) {

   String distinguishedName = "ov1 = " + direction

     + ", ov2 = " + division

     + ", ov3 = " + department

     + ", ov4 = " + service

     + ", cn = " + firstName + lastName;
```

```
    CompanyBEmployees companyBEmployees = new CompanyBEmployees();

    return companyBEmployees.getEmployee(distinguishedName);

  }

}
```

Implementation in Ruby

7.4.2 Ruby

```ruby
class Adaptee
  def specific_request
    # do something
  end
end

class Adapter
  def initialize(adaptee)
    @adaptee = adaptee
  end

  def request
    @adaptee.specific_request
  end
end

client = Adapter.new(Adaptee.new)
client.request
```

Implementation in Python

```python
class Adaptee:
    def specific_request(self):
        return 'Adaptee'

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

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

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

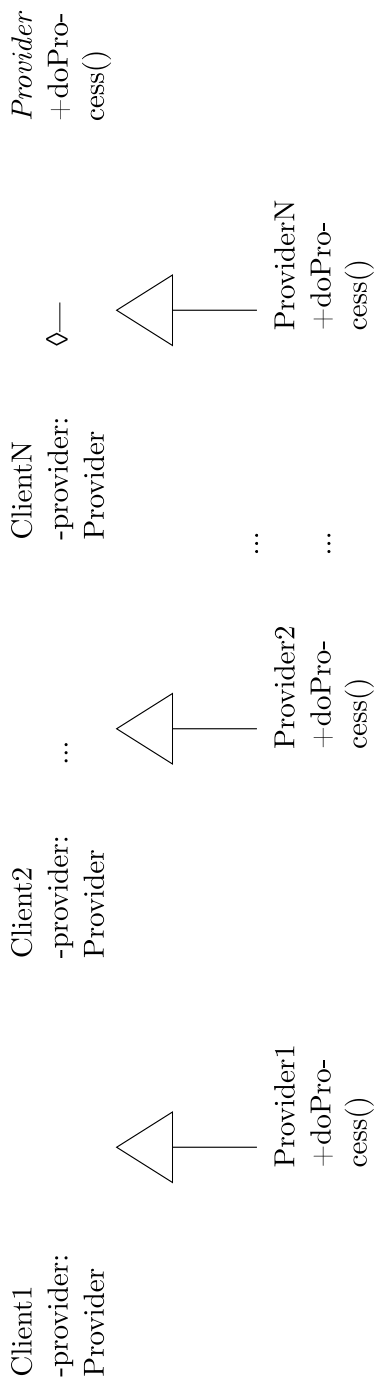
7.4.3 Class Adapter

Implementation in Python

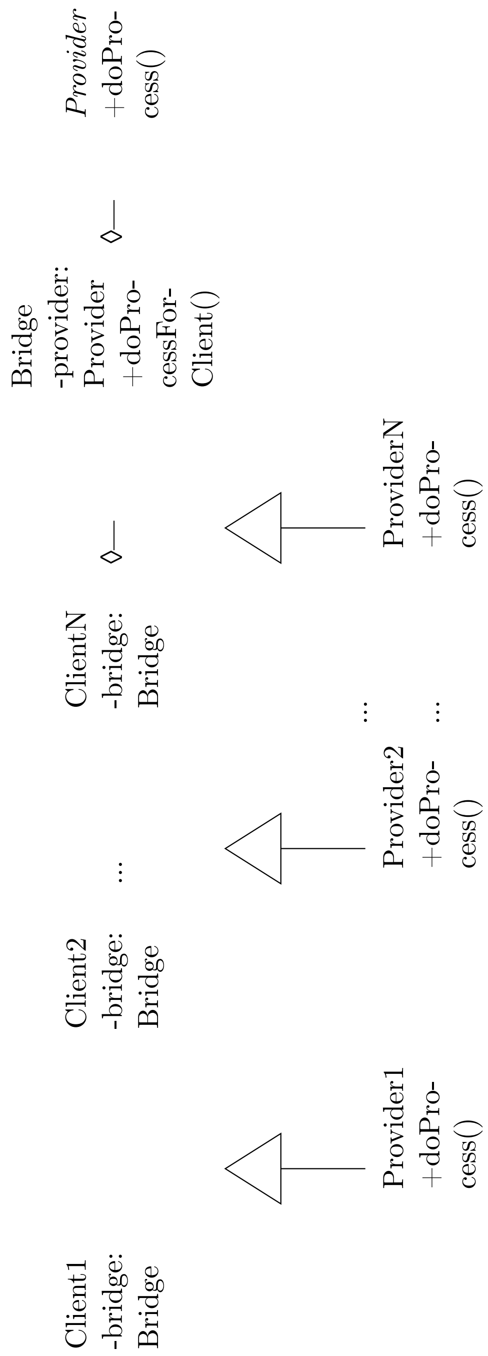
```
class Adaptee1:
    def __init__(self, *args, **kw):
        pass

    def specific_request(self):
        return 'Adaptee1'

class Adaptee2:
    def __init__(self, *args, **kw):
        pass

    def specific_request(self):
        return 'Adaptee2'

class Adapter(Adaptee1, Adaptee2):
    def __init__(self, *args, **kw):
        Adaptee1.__init__(self, *args, **kw)
        Adaptee2.__init__(self, *args, **kw)

    def request(self):
        return Adaptee1.specific_request(self), Adaptee2.specific_request(self)

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

8 Bridge

Bridge pattern is useful when a code often changes for an implementation as well as for a use of code. In your application, you should have provider classes and client classes:


Client1
-provider: Provider
Client2
-provider: Provider
...
ClientN
-provider: Provider
Provider
+doProcess()
Provider1
+doProcess()
Provider2
+doProcess()
...
...
ProviderN
+doProcess()

Each client class can interact with each provider class. However, if the implementation changes, the method signatures of the Provider interface may change and all the client classes have to change. In the same way, if the client classes need a new interface, we need to rewrite all the providers. The solution is to add a bridge, that is to say a class that will be called by the clients, that contains a reference to the providers and forward the client call to the providers.


Client1 -bridge: Bridge
Client2 -bridge: Bridge
...
ClientN -bridge: Bridge
Bridge -provider: Provider +doProcessForClient()
Provider +doProcess()
Provider1 +doProcess()
Provider2 +doProcess()
...
...
ProviderN +doProcess()

In the future, the two interfaces (client/bridge and bridge/provider) may change independently and the bridge may trans-code the call order.

8.1 Examples

It's hard to find an example in a library as this pattern is designed for versatile specifications and a library does not change constantly between two versions.

8.2 Cost

The cost of this pattern is the same as the adapter. It can be planned at design time but the best way to decide to add it is the experience feedback. Implement it when you have frequently changed an interface in a short time.

8.2.1 Creation

Its implementation is easy but can be expensive. It depends on the complexity of the interface. The more you have methods, the more it's expensive.

8.2.2 Maintenance

If you always update the client/bridge interface and the bridge/provider interface the same way, it would be more expensive than if you do not implement the design pattern.

8.2.3 Removal

This pattern can be easily removed as automatic refactoring operations can easily remove its existence.

8.3 Advices

- Put the *bridge* term in the name of the bridge class to indicate the use of the pattern to the other developers.

8.4 Implementation

Implementation in Java

The following Java[1] (SE 6) program illustrates the bridge pattern.

1 `http://en.wikibooks.org/wiki/Java_Programming`

```
/**
 * 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
 */
abstract class Shape {
   protected DrawingAPI drawingAPI;

   protected Shape(DrawingAPI drawingAPI) {
      this.drawingAPI = drawingAPI;
```

```
   }

   public abstract void draw();                              // low-level

   public abstract void resizeByPercentage(double pct);     // high-level

}

/**

 * Refined Abstraction

 */

class CircleShape extends Shape {

   private double x, y, radius;

   public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI)
 {

      super(drawingAPI);

      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) {
```

```
        Shape[] shapes = new Shape[] {

            new CircleShape(1, 2, 3, new DrawingAPI1()),

            new CircleShape(5, 7, 11, new DrawingAPI2()),

        };



        for (Shape shape : shapes) {

            shape.resizeByPercentage(2.5);

            shape.draw();

        }

    }

}
```

It will output:

```
API1.circle at 1.000000:2.000000 radius 7.5000000
API2.circle at 5.000000:7.000000 radius 27.500000
```

Implementation in PHP

```
interface DrawingAPI {
    function drawCircle($dX, $dY, $dRadius);
}

class DrawingAPI1 implements DrawingAPI {

    public function drawCircle($dX, $dY, $dRadius) {
        echo "API1.circle at ".$dX.":".$dY." radius ".$dRadius."br/";
    }
}

class DrawingAPI2 implements DrawingAPI {
```

```
    public function drawCircle($dX, $dY, $dRadius) {
        echo "API2.circle at ".$dX.":".$dY." radius ".$dRadius."br/";
    }
}

abstract class Shape {

    protected $oDrawingAPI;

    public abstract function draw();
    public abstract function resizeByPercentage($dPct);

    protected function __construct(DrawingAPI $oDrawingAPI) {
        $this-oDrawingAPI = $oDrawingAPI;
    }
}

class CircleShape extends Shape {

    private $dX;
    private $dY;
    private $dRadius;

    public function __construct(
            $dX, $dY,
            $dRadius,
            DrawingAPI $oDrawingAPI
    ) {
        parent::__construct($oDrawingAPI);
        $this-dX = $dX;
        $this-dY = $dY;
        $this-dRadius = $dRadius;
    }

    public function draw() {
        $this-oDrawingAPI-drawCircle(
                $this-dX,
                $this-dY,
                $this-dRadius
        );
    }

    public function resizeByPercentage($dPct) {
        $this-dRadius *= $dPct;
    }
}

class Tester {

    public static function main()  {
        $aShapes = array(
            new CircleShape(1, 3, 7,  new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2()),
        );

        foreach ($aShapes as $shapes) {
            $shapes-resizeByPercentage(2.5);
            $shapes-draw();
        }
    }
}

Tester::main();
```

Output:

```
API1.circle at 1:3 radius 17.5
API2.circle at 5:7 radius 27.5
```

Implementation in C#

8.4.1 C#

The following C#[2] 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;
   }
```

2 http://en.wikibooks.org/wiki/C_Sharp_Programming

```
    // 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();
        }
    }
}
```

8.4.2 C# using generics

The following C#[3] 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)
```

3 http://en.wikibooks.org/wiki/C_Sharp_Programming

```
}

/** "Refined Abstraction" */
class CircleShapeT : IShape
   where T : struct, IDrawingAPI
{
   private double x, y, radius;
   private 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 CircleShapeDrawingAPI1(1, 2, 3);
       shapes[1] = new CircleShapeDrawingAPI2(5, 7, 11);

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

Implementation in Python

The following Python[4] program illustrates the "shape" example given above and will output:

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

```python
# Implementor
class DrawingAPI:
    def drawCircle(x, y, radius):
        pass

# ConcreteImplementor 1/2
class DrawingAPI1(DrawingAPI):
    def drawCircle(self, x, y, radius):
        print "API1.circle at %f:%f radius %f" % (x, y, radius)

# ConcreteImplementor 2/2
class DrawingAPI2(DrawingAPI):
    def drawCircle(self, x, y, radius):
        print "API2.circle at %f:%f radius %f" % (x, y, radius)

# Abstraction
class Shape:
    # low-level
    def draw(self):
        pass
```

4 http://en.wikibooks.org/wiki/Python_Programming

```
    # high-level
    def resizeByPercentage(self, pct):
        pass

# Refined Abstraction
class CircleShape(Shape):
    def __init__(self, x, y, radius, drawingAPI):
        self.__x = x
        self.__y = y
        self.__radius = radius
        self.__drawingAPI = drawingAPI

    # low-level i.e. Implementation specific
    def draw(self):
        self.__drawingAPI.drawCircle(self.__x, self.__y, self.__radius)

    # high-level i.e. Abstraction specific
    def resizeByPercentage(self, pct):
        self.__radius *= pct

def main():
    shapes = [
        CircleShape(1, 2, 3, DrawingAPI1()),
        CircleShape(5, 7, 11, DrawingAPI2())
    ]

    for shape in shapes:
        shape.resizeByPercentage(2.5)
        shape.draw()

if __name__ == "__main__":
    main()
```

Implementation in Ruby

An example in Ruby[5].

```
class Abstraction
  def initialize(implementor)
    @implementor = implementor
  end

  def operation
    raise 'Implementor object does not respond to the operation method' unless
 @implementor.respond_to?(:operation)
    @implementor.operation
  end
end

class RefinedAbstraction  Abstraction
  def operation
    puts 'Starting operation... '
    super
  end
end

class Implementor
  def operation
    puts 'Doing neccessary stuff'
  end
end

class ConcreteImplementorA  Implementor
```

5 http://en.wikibooks.org/wiki/Ruby_Programming

```
  def operation
    super
    puts 'Doing additional stuff'
  end
end

class ConcreteImplementorB  Implementor
  def operation
    super
    puts 'Doing other additional stuff'
  end
end

normal_with_a = Abstraction.new(ConcreteImplementorA.new)
normal_with_a.operation
# Doing neccessary stuff
# Doing additional stuff

normal_with_b = Abstraction.new(ConcreteImplementorB.new)
normal_with_b.operation
# Doing neccessary stuff
# Doing other additional stuff

refined_with_a = RefinedAbstraction.new(ConcreteImplementorA.new)
refined_with_a.operation
# Starting operation...
# Doing neccessary stuff
# Doing additional stuff

refined_with_b = RefinedAbstraction.new(ConcreteImplementorB.new)
refined_with_b.operation
# Starting operation...
# Doing neccessary stuff
# Doing other additional stuff
```

Implementation in Scala

A Scala[6] implementation of the Java drawing example with the same output.

```
  /** "Implementor" */
  trait DrawingAPI {
    def drawCircle(x:Double, y:Double, radius:Double)
  }

  /** "ConcreteImplementor" 1/2 */
  class DrawingAPI1 extends DrawingAPI {
    def drawCircle(x:Double, y:Double, radius:Double) {
      printf("API1.circle at %f:%f radius %f\n", x, y, radius)
    }
  }

  /** "ConcreteImplementor" 2/2 */
  class DrawingAPI2 extends DrawingAPI {
    def drawCircle(x:Double, y:Double, radius:Double) {
      printf("API2.circle at %f:%f radius %f\n", x, y, radius)
    }
  }

  /** "Abstraction" */
  trait Shape {
     def draw()                             // low-level
     def resizeByPercentage(pct:Double)     // high-level
  }
```

6 http://en.wikibooks.org/wiki/Scala

```
  /** "Refined Abstraction" */
  class CircleShape(var x:Double, var y:Double,
    var radius:Double, val drawingAPI:DrawingAPI) extends Shape {

    // low-level i.e. Implementation specific
    def draw() = drawingAPI.drawCircle(x, y, radius)

    // high-level i.e. Abstraction specific
    def resizeByPercentage(pct:Double) = radius *= pct
  }

  /** "Client" */
  val shapes = List(
    new CircleShape(1, 2, 3, new DrawingAPI1),
    new CircleShape(5, 7, 11, new DrawingAPI2)
  )

  shapes foreach { shape =
    shape.resizeByPercentage(2.5)
    shape.draw()
  }
```

Implementation in D

An example in D[7].

```
import std.stdio;

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

/** "ConcreteImplementor" 1/2 */
class DrawingAPI1: DrawingAPI {
    void drawCircle(double x, double y, double radius) {
        writefln("\nAPI1.circle at %f:%f radius %f", x, y, radius);
    }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2: DrawingAPI {
    void drawCircle(double x, double y, double radius) {
        writefln("\nAPI2.circle at %f:%f radius %f", x, y, radius);
    }
}

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

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

7 http://en.wikibooks.org/wiki/D_Programming

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

int main(string[] argv) {
    auto api1 = new DrawingAPI1();
    auto api2 = new DrawingAPI2();

    auto c1 = new CircleShape(1, 2, 3, api1);
    auto c2 = new CircleShape(5, 7, 11, api2);

    Shape[4] shapes;
    shapes[0] = c1;
    shapes[1] = c2;

    shapes[0].resizeByPercentage(2.5);
    shapes[0].draw();
    shapes[1].resizeByPercentage(2.5);
    shapes[1].draw();

    return 0;
}
```

Implementation in Perl

This example in Perl[8] uses the MooseX::Declare[9] module.

```
# Implementor
role Drawing::API {
    requires 'draw_circle';
}

# Concrete Implementor 1
class Drawing::API::1 with Drawing::API {
    method draw_circle(Num $x, Num $y, Num $r) {
        printf "API1.circle at %f:%f radius %f\n", $x, $y, $r;
    }
}

# Concrete Implementor 2
class Drawing::API::2 with Drawing::API {
    method draw_circle(Num $x, Num $y, Num $r) {
        printf "API2.circle at %f:%f radius %f\n", $x, $y, $r;
    }
}

# Abstraction
role Shape {
    requires qw( draw resize );
}

# Refined Abstraction
class Shape::Circle with Shape {
```

8 http://en.wikibooks.org/wiki/Perl_Programming
9 http://search.cpan.org/search?query=MooseX%3A%3ADeclare&mode=all

```
    has $_  = ( is = 'rw', isa  = 'Any' ) for qw( x y r );
    has api = ( is = 'ro', does = 'Drawing::API' );

    method draw() {
        $self-api-draw_circle( $self-x, $self-y, $self-r );
    }

    method resize(Num $percentage) {
        $self-{r} *= $percentage;
    }
}

my @shapes = (
    Shape::Circle-new( x=1, y=2, r=3,  api = Drawing::API::1-new ),
    Shape::Circle-new( x=5, y=7, r=11, api = Drawing::API::2-new ),
)

$_-resize( 2.5 ) and $_-draw for @shapes;
```

9 Composite

The composite design pattern reduces the cost of an implementation that handles data represented as a tree. When an application does a process on a tree, usually the process has to handle the iteration on the components, the move on the tree and has to process the nodes and the leafs separately. All of this creates a big amount of code. Suppose that you have to handle a file system repertory. Each folders can contain files or folders. To handle this, you have an array of items that can be file or folder. The files have a name and the folders are arrays. Now you have to implement a file search operation on the whole folder tree. The pseudo-code should look like this:

```
method searchFilesInFolders(rootFolder, searchedFileName) is
    input: a list of the content of the rootFolder.
   input: the searchedFileName that should be found in the folders.
   output: the list of encountered files.

  Empty the foundFiles list
  Empty the parentFolders list
  Empty the parentIndices list
  currentFolder := rootFolder
  currentIndex := 0
  Add rootFolder to parentFolders

  while parentFolders is not empty do
    if currentIndex is out of currentFolder then
        currentFolder := last item of parentFolders
        Remove the last item of parentFolders
        currentIndex := last item of parentIndices + 1
        Remove the last item of parentIndices

    else if the item at the currentIndex of the currentFolder is a folder then
      currentFolder := the folder
      Add currentFolder to parentFolders
      Add currentIndex to parentIndices
      currentIndex := 0

    otherwise
      if the name of the file is equal to the searchedFileName then
        Add the file to foundFiles
      Increment currentIndex

  Return the foundFiles
```

In the previous code, each iteration of the same while loop is a process of one node or leaf. At the end of the process, the code move to the position of the next node or leaf to process. There are three branches in the loop. The first branch is true when we have processed all the children of the node and it moves to the parent, the second goes into a child node and the last process a leaf (i.e. a file). The memory of the location should be stored to go back in the tree. The problem in this implementation is that it is hardly readable and the process of the folders and the files is completely separate. This code is heavy to maintain and you have to think to the whole tree each moment. The folders and the files should be called the same way so they should be objects that implements the same interface.

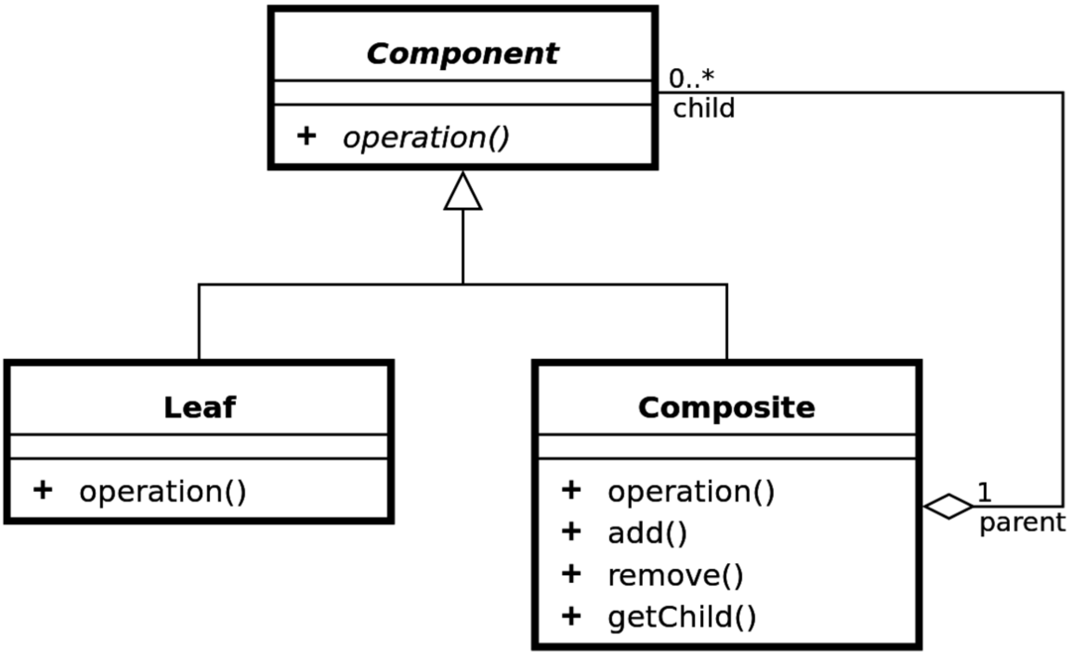


Figure 15 Composite pattern in UML.

Component

- is the abstraction for all components, including composite ones.
- declares the interface for objects in the composition.
- (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.

So now the implementation is rather like this:

interface FileSystemComponent **is**
  method searchFilesInFolders(searchedFileName) **is**
    input: the *searchedFileName* that should be found in the folders.
    output: the list of encountered files.

```
class File implementing FileSystemComponent is
 method searchFilesInFolders(searchedFileName) is
    input: the searchedFileName that should be found in the folders.
    output: the list of encountered files.

  if the name of the file is equal to the searchedFileName then
    Empty the foundFiles list
    Add the file to foundFiles
    Return the foundFiles

  otherwise
    Return an empty list


class Folder implementing FileSystemComponent is
 field children is
  The list of the direct children.

 method searchFilesInFolders(searchedFileName) is
    input: the searchedFileName that should be found in the folders.
    output: the list of encountered files.

  Empty the foundFiles list

  for each child in children
    Call searchFilesInFolders(searchedFileName) on the child
    Add the result to foundFiles

  Return the foundFiles
```

9.1 Examples

The best example of use of this pattern is the Graphical User Interface. The widgets of the interface are organized in a tree and the operations (resizing, repainting...) on all the widgets are processed using the composite design pattern.

9.2 Cost

This pattern is one of the less expensive patterns. You can implement it each time you have to handle a tree of data without worrying. There is no bad usage of this pattern. The cost of the pattern is only to handle the children of a composite but this cost would be required and more expensive without the design pattern.

9.2.1 Creation

You have to create an almost empty interface and implement the management of the composite children. This cost is very low.

9.2.2 Maintenance

You can't get caught in the system. The only relatively expensive situation occurs when you have to often change the operations applied to the whole data tree.

9.2.3 Removal

You should remove the pattern when you remove the data tree. So you just remove all in once. This cost is very low.

9.3 Advices

- Put the *composite* and *component* terms in the name of the classes to indicate the use of the pattern to the other developers.

9.4 Implementations

Various examples of the composite pattern.

Implementation in Java

The following example, written in Java[1], 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 | Graphic GraphicList
```

It could be extended to implement several other shapes (rectangle, etc.) and methods (translate[2], etc.).

```
/** "Component" */

interface Graphic {

    // Prints the graphic.
```

1 http://en.wikibooks.org/wiki/Subject:Java_programming_language
2 http://en.wikibooks.org//en.wikipedia.org/wiki/translation_(geometry)

```
    public void print();

}



/** "Composite" */

import java.util.List;

import java.util.ArrayList;




class CompositeGraphic implements Graphic {




    // Collection of child graphics.

    private ListGraphic mChildGraphics = new ArrayListGraphic();




    // 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();

    }

}
```

Implementation in C#

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

namespace Composite
{
    class Program
    {
        interface IGraphic
        {
            void Print();
        }

        class CompositeGraphic : IGraphic
        {
            private ListIGraphic child = new ListIGraphic();

            public CompositeGraphic(IEnumerableIGraphic collection)
            {
                child.AddRange(collection);
            }

            public void Print()
            {
                foreach(IGraphic g in child)
                {
                    g.Print();
                }
            }
        }

        class Ellipse : IGraphic
        {
            public void Print()
            {
                Console.WriteLine("Ellipse");
            }
        }

        static void Main(string[] args)
        {
            new CompositeGraphic(new IGraphic[]
                {
                    new CompositeGraphic(new IGraphic[]
                        {
                            new Ellipse(),
                            new Ellipse(),
                            new Ellipse()
                        }),
                    new CompositeGraphic(new IGraphic[]
                        {
                            new Ellipse()
                        })
                }).Print();
        }
    }
}
```

Implementation in Common Lisp

The following example, written in Common Lisp[3], and translated directly from the Java example below it, implements a method named *print-graphic*, which can be used on either an *ellipse*, or a list whose elements are either lists or *ellipses*.

```
(defstruct ellipse) ;; An empty struct.

;; For the method definitions, "object" is the variable,
;; and the following word is the type.

(defmethod print-graphic ((object null))
  NIL)

(defmethod print-graphic ((object cons))
  (print-graphic (first object))
  (print-graphic (rest object)))

(defmethod print-graphic ((object ellipse))
  (print 'ELLIPSE))

(let* ((ellipse-1 (make-ellipse))
       (ellipse-2 (make-ellipse))
       (ellipse-3 (make-ellipse))
       (ellipse-4 (make-ellipse)))

  (print-graphic (cons (list ellipse-1 (list ellipse-2 ellipse-3)) ellipse-4)))
```

Implementation in PHP

```
?php

/** "Component" */
interface Graphic
{
    /**
     * Prints the graphic
     *
     * @return void
     */
    public function printOut();
}

/**
 * "Composite" - Collection of graphical components
 */
class CompositeGraphic implements Graphic
{
    /**
     * Collection of child graphics
     *
     * @var array
     */
    private $childGraphics = array();

    /**
     * Prints the graphic
     *
     * @return void
     */
    public function printOut()
    {
```

3 http://en.wikibooks.org/wiki/Common_Lisp

```
        foreach ($this-childGraphics as $graphic) {
            $graphic-printOut();
        }
    }

    /**
     * Adds the graphic to the composition
     *
     * @param Graphic $graphic Graphical element
     *
     * @return void
     */
    public function add(Graphic $graphic)
    {
        $this-childGraphics[] = $graphic;
    }

    /**
     * Removes the graphic from the composition
     *
     * @param Graphic $graphic Graphical element
     *
     * @return void
     */
    public function remove(Graphic $graphic)
    {
        if (in_array($graphic, $this-childGraphics)) {
            unset($this-childGraphics[array_search($graphic,
 $this-childGraphics)]);
        }
    }
}

/** "Leaf" */
class Ellipse implements Graphic
{
    /**
     * Prints the graphic
     *
     * @return void
     */
    public function printOut()
    {
        echo "Ellipse";
    }
}

/** Client */

//Initialize four ellipses
$ellipse1 = new Ellipse();
$ellipse2 = new Ellipse();
$ellipse3 = new Ellipse();
$ellipse4 = new Ellipse();

//Initialize three composite graphics
$graphic = new CompositeGraphic();
$graphic1 = new 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-printOut();
```

Implementation in Python

```
class Component(object):
    def __init__(self, *args, **kw):
        pass

    def component_function(self): pass

class Leaf(Component):
    def __init__(self, *args, **kw):
        Component.__init__(self, *args, **kw)

    def component_function(self):
        print "some function"

class Composite(Component):
    def __init__(self, *args, **kw):
        Component.__init__(self, *args, **kw)
        self.children = []

    def append_child(self, child):
        self.children.append(child)

    def remove_child(self, child):
        self.children.remove(child)

    def component_function(self):
        map(lambda x: x.component_function(), self.children)

c = Composite()
l = Leaf()
l_two = Leaf()
c.append_child(l)
c.append_child(l_two)
c.component_function()
```

Implementation in Ruby

```
module Component
  def do_something
    raise NotImplementedError
  end
end

class Leaf
  include Component
  def do_something
    puts "Hello"
  end
end

class Composite
  include Component
  attr_accessor :children
  def initialize
    self.children = []
  end
  def do_something
    children.each {|c| c.do_something}
```

```
  end
  def append_child(child)
    children  child
  end
  def remove_child(child)
    children.delete child
  end
end

composite = Composite.new
leaf_one = Leaf.new
leaf_two = Leaf.new
composite.append_child(leaf_one)
composite.append_child(leaf_two)
composite.do_something
```

10 Print version

Various examples of the decorator pattern.

Implementation in C#

This example illustrates a simple extension method for a bool type.

```
using System;

static class BooleanExtensionMethodSample
{
    public static void Main()
    {
        bool yes = true;
        bool no = false;
        // Toggle the booleans! yes should return false and no should return
 true.
        Console.WriteLine(yes.Toggle());
        Console.WriteLine(no.Toggle());
    }
    // The extension method that adds Toggle to bool.
    public static bool Toggle(this bool target)
    {
        // Evaluate the input and then return the opposite value.
        if (target)
            return false;
        else
            return true; // Satisfy the compiler in case of a null value.
    }
}
```

Implementation in Java

10.1 First Example (window/scrolling scenario)

The following Java example illustrates the use of decorators using the window/scrolling scenario.

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

// extension of a simple Window without any scrollbars
class SimpleWindow extends 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 extends Window
abstract class WindowDecorator extends Window {
    protected Window decoratedWindow; // the Window being decorated

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
    public void draw() {
        decoratedWindow.draw(); //delegation
    }
    public String getDescription() {
        return decoratedWindow.getDescription(); //delegation
    }
}

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

    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }

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

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

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

    @Override
    public void draw() {
        super.draw();
        drawHorizontalScrollBar();
    }

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

    @Override
    public String getDescription() {
        return super.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.

10.2 Second Example (coffee making scenario)

The next Java example illustrates the use of decorators using coffee making scenario. In this example, the scenario only includes cost and ingredients.

```
// The abstract Coffee class defines the functionality of Coffee implemented by
 decorator
public abstract class Coffee {
    public abstract double getCost(); // returns the cost of the coffee
    public abstract String getIngredients(); // returns the ingredients of the
 coffee
}

// extension of a simple coffee without any extra ingredients
public class SimpleCoffee extends Coffee {
    public double getCost() {
        return 1;
    }

    public String getIngredients() {
        return "Coffee";
    }
}
```

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

```
// abstract decorator class - note that it extends Coffee abstract class
public abstract class CoffeeDecorator extends Coffee {
    protected final Coffee decoratedCoffee;
    protected String ingredientSeparator = ", ";

    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }
```

```
    public double getCost() { // implementing methods of the abstract class
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}

// Decorator Milk that mixes milk with coffee
// note it extends CoffeeDecorator
class Milk extends CoffeeDecorator {
    public Milk(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() { // overriding methods defined in the abstract
 superclass
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Milk";
    }
}

// Decorator Whip that mixes whip with coffee
// note it extends CoffeeDecorator
class Whip extends CoffeeDecorator {
    public Whip(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() {
        return super.getCost() + 0.7;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Whip";
    }
}

// Decorator Sprinkles that mixes sprinkles with coffee
// note it extends CoffeeDecorator
class Sprinkles extends CoffeeDecorator {
    public Sprinkles(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    public double getCost() {
        return super.getCost() + 0.2;
    }

    public String getIngredients() {
        return super.getIngredients() + ingredientSeparator + "Sprinkles";
    }
}
```

Here's a test program that creates a `Coffee` instance which is fully decorated (i.e., with milk, whip, sprinkles), and calculate cost of coffee and prints its ingredients:

```
public class Main {

    public static final void main(String[] args) {
```

```
Coffee c = new SimpleCoffee();
System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());

c = new Milk(c);
System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());

c = new Sprinkles(c);
System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());

c = new Whip(c);
System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());

// Note that you can also stack more than one decorator of the same type
c = new Sprinkles(c);
System.out.println("Cost: " + c.getCost() + "; Ingredients: " +
 c.getIngredients());
    }

}
```

The output of this program is given below:

```
Cost: 1.0 Ingredient: Coffee


Cost: 1.5 Ingredient: Coffee, Milk


Cost: 1.7 Ingredient: Coffee, Milk, Sprinkles


Cost: 2.4 Ingredient: Coffee, Milk, Sprinkles, Whip
```

Implementation in C++

10.3 Coffee making scenario

```
# include iostream
# include string

// The abstract coffee class
class Coffee
{
public:
    virtual double getCost() = 0;
    virtual std::string getIngredient() = 0;
};

// Plain coffee without ingredient
class SimpleCoffee:public Coffee
{
private:
    double cost;
    std::string ingredient;
```

```
public:
    SimpleCoffee()
    {
        cost = 1;
        ingredient = std::string("Coffee");
    }
    double getCost()
    {
        return cost;
    }
    std::string getIngredient()
    {
        return ingredient;
    }
};

// Abstract decorator class
class CoffeeDecorator:public Coffee
{
protected:
    Coffee  decoratedCoffee;
public:
    CoffeeDecorator(Coffee  decoratedCoffee):decoratedCoffee(decoratedCoffee){}
};

// Milk Decorator
class Milk:public CoffeeDecorator
{
private:
    double cost;

public:
    Milk(Coffee  decoratedCoffee):CoffeeDecorator(decoratedCoffee)
    {
        cost = 0.5;
    }
    double getCost()
    {
        return cost + decoratedCoffee.getCost();
    }
    std::string getIngredient()
    {
        return "Milk "+decoratedCoffee.getIngredient();
    }
};

// Whip decorator
class Whip:public CoffeeDecorator
{
private:
    double cost;

public:
    Whip(Coffee  decoratedCoffee):CoffeeDecorator(decoratedCoffee)
    {
        cost = 0.7;
    }
    double getCost()
    {
        return cost + decoratedCoffee.getCost();
    }
    std::string getIngredient()
    {
        return "Whip "+decoratedCoffee.getIngredient();
    }
};

// Sprinkles decorator
```

```
class Sprinkles:public CoffeeDecorator
{
private:
    double cost;

public:
    Sprinkles(Coffee  decoratedCoffee):CoffeeDecorator(decoratedCoffee)
    {
        cost = 0.6;
    }
    double getCost()
    {
        return cost + decoratedCoffee.getCost();
    }
    std::string getIngredient()
    {
        return "Sprinkles "+decoratedCoffee.getIngredient();
    }
};

// Here's a test
int main()
{
    Coffee* sample;
    sample = new SimpleCoffee();
    sample = new Milk(*sample);
    sample = new Whip(*sample);
    std::cout  sample-getIngredient()  std::endl;
    std::cout  "Cost: "  sample-getCost()  std::endl;
}
```

The output of this program is given below:

```
Whip Milk Coffee


Cost: 2.2
```

Implementation in JavaScript

```
// 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());
```

Implementation in Python

10.4 Window System

```
# the Window base class
class Window(object):
    def draw(self, device):
        device.append('flat window')
    def info(self):
        pass
# The decorator pattern approch
class WindowDecorator:
    def __init__(self, w):
        self.window = w
    def draw(self, device):
        self.window.draw(device)
    def info(self):
        self.window.info()

class BorderDecorator(WindowDecorator):
    def draw(self, device):
        self.window.draw(device)
        device.append('borders')

class ScrollDecorator(WindowDecorator):
    def draw(self, device):
        self.window.draw(device)
        device.append('scroll bars')

def test_deco():
    # The way of using the decorator classes
    w = ScrollDecorator(BorderDecorator(Window()))
    dev = []
    w.draw(dev)
    print dev
test_deco()
```

10.4.1 Difference between subclass method and decorator pattern

```
# The subclass approch
class BorderedWindow(Window):
    def draw(self, device):
        super(BorderedWindow, self).draw(device)
```

```
        device.append('borders')

class ScrolledWindow(Window):
    def draw(self, device):
        super(ScrolledWindow, self).draw(device)
        device.append('scroll bars')

# combine the functionalities using multiple inheritance.
class MyWindow(ScrolledWindow, BorderedWindow, Window):
    pass

def test_muli():
    w = MyWindow()
    dev = []
    w.draw(dev)
    print dev

def test_muli2():
    # note that python can create a class on the fly.
    MyWindow = type('MyWindow', (ScrolledWindow, BorderedWindow, Window), {})
    w = MyWindow()
    dev = []
    w.draw(dev)
    print dev

test_muli()
test_muli2()
```

10.5 Dynamic languages

The decorator pattern can also be implemented in dynamic languages either with interfaces or with traditional OOP inheritance.

10.6 External links

Java Design Patterns tutorial[1] History of Design Patterns[2]

1 http://javadesign-patterns.blogspot.com
2 http://c2.com/cgi/wiki?HistoryOfPatterns

11 Print version

One class has a method that performs a complex process calling several other classes.

Implementation in Java

This is an abstract example of how a client ("you") interacts with a facade (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) { ... }
}


/* Facade */

class Computer {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public Computer() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}


/* Client */

class You {
```

```
    public static void main(String[] args) {
        Computer facade = new Computer();
        facade.start();
    }
}
```

Implementation in C#

```
using System;

namespace Facade
{
        public class CPU
        {
                public void Freeze() { }
                public void Jump(long addr) { }
                public void Execute() { }
        }

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

        public class HardDrive
        {
                public byte[] Read(long lba, int size) { return null; }
        }

        public class Computer
        {
                var cpu = new CPU();
                var memory = new Memory();
                var hardDrive = new HardDrive();

                public void StartComputer()
                {
                        cpu.Freeze();
                        memory.Load(0x22, hardDrive.Read(0x66, 0x99));
                        cpu.Jump(0x44);
                        cpu.Execute();
                }
        }

        public class SomeClass
        {
            public static void Main(string[] args)
            {
                var facade = new Computer();
                facade.StartComputer();
            }
        }
}
```

Implementation in Ruby

```
# Complex parts
class CPU
  def freeze; puts 'CPU: freeze'; end
  def jump(position); puts "CPU: jump to #{position}"; end
  def execute; puts 'CPU: execute'; end
end

class Memory
```

```
  def load(position, data)
    puts "Memory: load #{data} at #{position}"
  end
end

class HardDrive
  def read(lba, size)
    puts "HardDrive: read sector #{lba} (#{size} bytes)"
    return 'hdd data'
  end
end

# Facade
class Computer
  BOOT_ADDRESS = 0
  BOOT_SECTOR = 0
  SECTOR_SIZE = 512

  def initialize
    @cpu = CPU.new
    @memory = Memory.new
    @hard_drive = HardDrive.new
  end

  def start_computer
    @cpu.freeze
    @memory.load(BOOT_ADDRESS, @hard_drive.read(BOOT_SECTOR, SECTOR_SIZE))
    @cpu.jump(BOOT_ADDRESS)
    @cpu.execute
  end
end

# Client
facade = Computer.new
facade.start_computer
```

Implementation in Python

```
# Complex parts
class CPU:
    def freeze(self): pass
    def jump(self, position): pass
    def execute(self): pass

class Memory:
    def load(self, position, data): pass

class HardDrive:
    def read(self, lba, size): pass

# Facade
class Computer:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.hard_drive = HardDrive()

    def start_computer(self):
        self.cpu.freeze()
        self.memory.load(0, self.hard_drive.read(0, 1024))
        self.cpu.jump(10)
        self.cpu.execute()

# Client
if __name__ == '__main__':
```

```
    facade = Computer()
    facade.start_computer()
```

Implementation in PHP

```
/* Complex parts */
class CPU
{
    public function freeze() { /* ... */ }
    public function jump( $position ) { /* ... */ }
    public function execute() { /* ... */ }

}

class Memory
{
    public function load( $position, $data ) { /* ... */ }
}

class HardDrive
{
    public function read( $lba, $size ) { /* ... */ }
}

/* Facade */
class Computer
{
    protected $cpu = null;
    protected $memory = null;
    protected $hardDrive = null;

    public function __construct()
    {
        $this-cpu = new CPU();
        $this-memory = new Memory();
        $this-hardDrive = new HardDrive();
    }

    public function startComputer()
    {
        $this-cpu-freeze();
        $this-memory-load( BOOT_ADDRESS, $this-hardDrive-read( BOOT_SECTOR,
 SECTOR_SIZE ) );
        $this-cpu-jump( BOOT_ADDRESS );
        $this-cpu-execute();
    }
}

/* Client */
$facade = new Computer();
$facade-startComputer();
```

Implementation in JavaScript

```
/* Complex parts */
var CPU = function () {};
CPU.prototype = {
  freeze: function () {
    console.log('CPU: freeze');
  },
  jump: function (position) {
    console.log('CPU: jump to ' + position);
  },
  execute: function () {
```

```
    console.log('CPU: execute');
  }
};

var Memory = function () {};
Memory.prototype = {
  load: function (position, data) {
    console.log('Memory: load "' + data + '" at ' + position);
  }
};

var HardDrive = function () {};
HardDrive.prototype = {
  read: function (lba, size) {
    console.log('HardDrive: read sector ' + lba + '(' + size + ' bytes)');
    return 'hdd data';
  }
};


/* Facade */
var Computer = function () {
  var cpu, memory, hardDrive;

  cpu = new CPU();
  memory = new Memory();
  hardDrive = new HardDrive();

  var constant = function (name) {
    var constants = {
      BOOT_ADDRESS: 0,
      BOOT_SECTOR: 0,
      SECTOR_SIZE: 512
    };

    return constants[name];
  };

  this.startComputer = function () {
    cpu.freeze();
    memory.load(constant('BOOT_ADDRESS'),
 hardDrive.read(constant('BOOT_SECTOR'), constant('SECTOR_SIZE')));
    cpu.jump(constant('BOOT_ADDRESS'));
    cpu.execute();
  }

};

/* Client */
var facade = new Computer();
facade.startComputer();
```

Implementation in ActionScript 3.0

```
/* Complex Parts */

/* CPU.as */
package
{
    public class CPU
    {
        public function freeze():void
        {
            trace("CPU::freeze");
        }
```

```
        public function jump(addr:Number):void
        {
            trace("CPU::jump to", String(addr));
        }

        public function execute():void
        {
            trace("CPU::execute");
        }
    }
}

/* Memory.as */
package
{
    import flash.utils.ByteArray;

    public class Memory
    {
        public function load(position:Number, data:ByteArray):void
        {
            trace("Memory::load position:", position, "data:", data);
        }
    }
}

/* HardDrive.as */
package
{
    import flash.utils.ByteArray;

    public class HardDrive
    {
        public function read(lba:Number, size:int):ByteArray
        {
            trace("HardDrive::read returning null");
            return null;
        }
    }
}

/* The Facade */
/* Computer.as */
package
{
    public class Computer
    {
        public static const BOOT_ADDRESS:Number = 0x22;
        public static const BOOT_SECTOR:Number = 0x66;
        public static const SECTOR_SIZE:int = 0x200;

        private var _cpu:CPU;
        private var _memory:Memory;
        private var _hardDrive:HardDrive;

        public function Computer()
        {
            _cpu = new CPU();
            _memory = new Memory();
            _hardDrive = new HardDrive();
        }

        public function startComputer():void
        {
            _cpu.freeze();
            _memory.load(BOOT_ADDRESS, _hardDrive.read(BOOT_SECTOR,
 SECTOR_SIZE));
            _cpu.jump(BOOT_ADDRESS);
```

```
            _cpu.execute();
        }
    }
}

/* Client.as : This is the application's Document class */
package
{
    import flash.display.MovieClip;

    public class Client extends MovieClip
    {
        private var _computer:Computer;

        public function Client()
        {
            _computer = new Computer();
            _computer.startComputer();
        }
    }
}
```

12 Print version

Shared information are stored once.

Implementation in Java

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

```
import java.lang.ref.WeakReference;
import java.util.WeakHashMap;
import java.util.Collections;
import java.util.EnumSet;
import java.util.Set;
import java.awt.Color;

public final class FontData {

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

    /**
     * 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 WeakHashMapFontData, WeakReferenceFontData
 FLY_WEIGHT_DATA =
        new WeakHashMapFontData, WeakReferenceFontData();
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final SetFontEffect effects;

    private FontData(int pointSize, String fontFace, Color color,
 EnumSetFontEffect 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) {
        EnumSetFontEffect 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);
```

```
        FontData result = null;
        // Retrieve previously created instance with the given values if it
 (still) exists
        WeakReferenceFontData ref = FLY_WEIGHT_DATA.get(data);
        if (ref != null) {
            result = ref.get();
        }

        // Store new font data instance if no matching instance exists
        if(result == null){
            FLY_WEIGHT_DATA.put(data, new WeakReferenceFontData (data));
            result = data;
        }

        // return the single immutable copy with the given values
        return result;
    }

    @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.
}
```

13 Print version

Control the access to an object. The example creates first an interface against which the pattern creates the classes. This interface contains only one method to display the image, called `displayImage()`, that has to be coded by all classes implementing it. The proxy class `ProxyImage` is running on another system than the real image class itself and can represent the real image `RealImage` over there. The image information is accessed from the disk. Using the proxy pattern, the code of the `ProxyImage` avoids multiple loading of the image, accessing it from the other system in a memory-saving manner.

Implementation in C#

```
using System;

namespace Proxy
{
    class Program
    {
        interface IImage
        {
            void Display();
        }

        class RealImage : IImage
        {
            public RealImage(string fileName)
            {
                FileName = fileName;
                LoadFromFile();
            }

            private void LoadFromFile()
            {
                Console.WriteLine("Loading " + FileName);
            }

            public String FileName { get; private set; }

            public void Display()
            {
                Console.WriteLine("Displaying " + FileName);
            }
        }

        class ProxyImage : IImage
        {
            public ProxyImage(string fileName)
            {
                FileName = fileName;
            }

            public String FileName { get; private set; }

            private IImage image;
```

```
            public void Display()
            {
                if (image == null)
                    image = new RealImage(FileName);
                image.Display();
            }
        }

        static void Main(string[] args)
        {
            IImage image = new ProxyImage("HiRes_Image");
            for (int i = 0; i  10; i++)
                image.Display();
        }
    }
}
```

The program's output is:

```
Loading HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
Displaying HiRes_Image
```

Implementation in Java

The following Java[1] example illustrates the "virtual proxy" pattern. The `ProxyImage` class is used to access a remote method.

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

//on System A
class RealImage implements Image {

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

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

1 http://en.wikibooks.org/wiki/Subject:Java_programming_language

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

}

//on System B
class ProxyImage implements Image {

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

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

}

class ProxyExample {

   /**
    * Test method
    */
   public static void main(String[] args) {
        final Image IMAGE1 = new ProxyImage("HiRes_10MB_Photo1");
        final Image IMAGE2 = new ProxyImage("HiRes_10MB_Photo2");

        IMAGE1.displayImage(); // loading necessary
        IMAGE1.displayImage(); // loading unnecessary
        IMAGE2.displayImage(); // loading necessary
        IMAGE2.displayImage(); // loading unnecessary
        IMAGE1.displayImage(); // loading unnecessary
    }

}
```

The program's output is:

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

14 Print version

Various examples of the Chain of responsibility pattern.

Implementation in 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. Note that this example should not be seen as a recommendation on how to write logging classes. Also, note that in a 'pure' implementation of the chain of responsibility pattern, 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.

```
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.err.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);
    }

}
```

The output is:

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

Below is another example of this pattern in Java. In this example we have different roles, each having a fix purchase power limit and a successor. Every time a user in a role receives a purchase request, when it's over his limit, he just passes that request to his successor. The PurchasePower abstract class with the abstract method processRequest.

```
abstract class PurchasePower {

    protected final double base = 500;
    protected PurchasePower successor;

    public void setSuccessor(PurchasePower successor) {
        this.successor = successor;
    }

    abstract public void processRequest(PurchaseRequest request);

}
```

Four implementations of the abstract class above: Manager, Director, Vice President, President

```
class ManagerPPower extends PurchasePower {

    private final double ALLOWABLE = 10 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount()  ALLOWABLE) {
            System.out.println("Manager will approve $"+ request.getAmount());
        }
        else if(successor != null) {
            successor.processRequest(request);
        }
    }

}
```

```
class DirectorPPower extends PurchasePower {

    private final double ALLOWABLE = 20 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount()  ALLOWABLE) {
            System.out.println("Director will approve $"+ request.getAmount());
        }
        else if(successor != null) {
            successor.processRequest(request);
        }
    }

}
```

```
class VicePresidentPPower extends PurchasePower {

    private final double ALLOWABLE = 40 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount()  ALLOWABLE) {
            System.out.println("Vice President will approve $" +
 request.getAmount());
        }
        else if(successor != null) {
            successor.processRequest(request);
```

```
        }
    }
}


class PresidentPPower extends PurchasePower {

    private final double ALLOWABLE = 60 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount()  ALLOWABLE) {
            System.out.println("President will approve $" +
 request.getAmount());
        }
        else {
            System.out.println( "Your request for $" + request.getAmount() + "
 needs a board meeting!");
        }
    }

}
```

The PurchaseRequest class with its Getter methods which keeps the request data in this example.

```
class PurchaseRequest {

    private int number;
    private double amount;
    private String purpose;

    public PurchaseRequest(int number, double amount, String purpose) {
        this.number = number;
        this.amount = amount;
        this.purpose = purpose;
    }

    public double getAmount() {
        return amount;
    }
    public void setAmount(double amt)  {
        amount = amt;
    }

    public String getPurpose() {
        return purpose;
    }
    public void setPurpose(String reason) {
        purpose = reason;
    }

    public int getNumber(){
        return number;
    }
    public void setNumber(int num) {
        number = num;
    }

}
```

And here a usage example, the successors are set like this: Manager -> Director -> Vice President -> President

```
class CheckAuthority {

    public static void main(String[] args) {
        ManagerPPower manager = new ManagerPPower();
        DirectorPPower director = new DirectorPPower();
        VicePresidentPPower vp = new VicePresidentPPower();
        PresidentPPower president = new PresidentPPower();
        manager.setSuccessor(director);
        director.setSuccessor(vp);
        vp.setSuccessor(president);

        //enter ctrl+c to kill.
        try{
            while (true) {
                System.out.println("Enter the amount to check who should approve
 your expenditure.");
                System.out.print("");
                double d = Double.parseDouble(new BufferedReader(new
 InputStreamReader(System.in)).readLine());
                manager.processRequest(new PurchaseRequest(0, d, "General"));
           }
        }
        catch(Exception e){
            System.exit(1);
        }
    }

}
```

Implementation in Python

```
class Car:
    def __init__(self):
        self.name = None
        self.km = 11100
        self.fuel = 5
        self.oil = 5

def handle_fuel(car):
    if car.fuel  10:
        print "added fuel"
        car.fuel = 100

def handle_km(car):
    if car.km  10000:
        print "made a car test."
        car.km = 0

def handle_oil(car):
    if car.oil  10:
        print "Added oil"
        car.oil = 100

class Garage:
    def __init__(self):
        self.handlers = []

    def add_handler(self, handler):
        self.handlers.append(handler)

    def handle_car(self, car):
        for handler in self.handlers:
            handler(car)
```

```
if __name__ == '__main__':
    handlers = [handle_fuel, handle_km, handle_oil]
    garage = Garage()

    for handle in handlers:
        garage.add_handler(handle)
    garage.handle_car(Car())
```

Implementation in Racket

```
#lang racket

; Define an automobile structure
(struct auto (fuel km oil) #:mutable #:transparent)

; Create an instance of an automobile
(define the-auto (auto 5 1500 7))

; Define handlers

(define (handle-fuel auto)
  (when ( (auto-fuel auto) 10)
    (begin (set-auto-fuel! auto 10)
           (printf "set fuel\n"))))

(define (handle-km auto)
  (when ( (auto-km auto) 10000)
    (begin (set-auto-km! auto 0)
           (printf "made a car test\n"))))

(define (handle-oil auto)
  (when ( (auto-oil auto) 10)
    (begin (set-auto-oil! auto (+ (auto-oil auto) 5))
           (printf "added oil\n"))))

; Apply each handler to the auto
(define (handle-auto handlers auto)
  (unless (null? handlers)
    (begin ((car handlers) auto)
           (handle-auto (cdr handlers) auto))))

(display the-auto)
(newline)

; Handle the auto
(handle-auto (list handle-fuel handle-km handle-oil) the-auto)

(display the-auto)
(newline)
```

Implementation in Perl

Java example about purchase power of various roles, using perl 5.10 and Moose.

```
use feature ':5.10';

package PurchasePower;
use Moose;

has successor = ( is = "rw", isa = "PurchasePower" );

sub processRequest {
    my ( $self, $req ) = @_;
    if ( $req-amount  $self-allowable ) {
```

```
        printf "%s will approve \$%.2f\n", $self-title, $req-amount;
    }
    elsif ( $self-successor ) {
        $self-successor-processRequest($req);
    }
    else {
        $self-no_match( $req );
    }
}

package ManagerPPower;
use Moose; extends "PurchasePower";
sub allowable {5000}
sub title     {"manager"}

package DirectorPPower;
use Moose; extends "PurchasePower";
sub allowable {10000}
sub title     {"director"}

package VicePresidentPPower;
use Moose; extends "PurchasePower";
sub allowable {20000}
sub title     {"vice-president"}

package PresidentPPower;
use Moose; extends "PurchasePower";
sub allowable {30000}
sub title     {"president"}

sub no_match {
    my ( $self, $req ) = @_;
    printf "your request for \$%.2f will need a board meeting\n", $req-amount;
}

package PurchaseRequest;
use Moose;

has number  = ( is = "rw", isa = "Int" );
has amount  = ( is = "rw", isa = "Num" );
has purpose = ( is = "rw", isa = "Str" );

package main;

my $manager   = new ManagerPPower;
my $director  = new DirectorPPower;
my $vp        = new VicePresidentPPower;
my $president = new PresidentPPower;

$manager-successor($director);
$director-successor($vp);
$vp-successor($president);

print "Enter the amount to check who should approve your expenditure.\n";
my $amount = readline;
$manager-processRequest(
    PurchaseRequest-new(
        number  = 0,
        amount  = $amount,
        purpose = "General"
    )
);
```

Perl example using perl 5.10 and Moose.

```
use feature ':5.10';
```

```
package Car;
use Moose;

has name = ( is = "rw", default = undef );
has km   = ( is = "rw", default = 11100 );
has fuel = ( is = "rw", default = 5 );
has oil  = ( is = "rw", default = 5 );

sub handle_fuel {
    my $self = shift;
    if ( $self-fuel  10 ) {
        say "added fuel";
        $self-fuel(100);
    }
}

sub handle_km {
    my $self = shift;
    if ( $self-km  10000 ) {
        say "made a car test";
        $self-km(0);
    }
}

sub handle_oil {
    my $self = shift;
    if ( $self-oil  10 ) {
        say "added oil";
        $self-oil(100);
    }
}

package Garage;
use Moose;

has handler = (
    is      = "ro",
    isa     = "ArrayRef[Str]",
    traits  = ['Array'],
    handles = { add_handler = "push", handlers = "elements" },
    default = sub { [] }
);

sub handle_car {
    my ($self, $car) = @_;
    $car-$_ for $self-handlers
}

package main;

my @handlers = qw( handle_fuel handle_km handle_oil );
my $garage   = Garage-new;
$garage-add_handler($_) for @handlers;
$garage-handle_car( Car-new );
```

Implementation in C#

```
using System;

  class MainApp
  {
    static void Main()
    {
      // Setup Chain of Responsibility
      Handler h1 = new ConcreteHandler1();
```

```
    Handler h2 = new ConcreteHandler2();
    Handler h3 = new ConcreteHandler3();
    h1.SetSuccessor(h2);
    h2.SetSuccessor(h3);

    // Generate and process request
    int[] requests = {2, 5, 14, 22, 18, 3, 27, 20};

    foreach (int request in requests)
    {
      h1.HandleRequest(request);
    }

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

// "Handler"
abstract class Handler
{
  protected Handler successor;

  public void SetSuccessor(Handler successor)
  {
    this.successor = successor;
  }

  public abstract void HandleRequest(int request);
}

// "ConcreteHandler1"
class ConcreteHandler1 : Handler
{
  public override void HandleRequest(int request)
  {
    if (request = 0  request  10)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}

// "ConcreteHandler2"
class ConcreteHandler2 : Handler
{
  public override void HandleRequest(int request)
  {
    if (request = 10  request  20)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}

// "ConcreteHandler3"
class ConcreteHandler3 : Handler
{
```

```
  public override void HandleRequest(int request)
  {
    if (request = 20  request  30)
    {
      Console.WriteLine("{0} handled request {1}",
        this.GetType().Name, request);
    }
    else if (successor != null)
    {
      successor.HandleRequest(request);
    }
  }
}
```

15 Print version

The command pattern is an Object behavioural pattern that decouples sender and receiver. It can also be thought as an object oriented equivalent of call back method. Call Back: It is a function that is registered to be called at later point of time based on user actions.

```
public interface Command {
  public int execute(int a, int b);
}
```

```
public class AddCommand implements Command {
  public int execute(int a, int b) {
     return a + b;
  }
}
```

```
public class MultCommand implements Command {
  public int execute(int a, int b) {
     return a * b;
  }
}
```

```
public class TestCommand {
  public static void main(String a[]) {
    Command add = new AddCommand();
    add.execute(1, 2); // returns 3
    Command multiply = new MultCommand();
    multiply.execute(2, 3); // returns 6
  }
}
```

In the above example, it can be noted that the command pattern decouples the object that invokes the operation from the ones having the knowledge to perform it.

15.1 Implementations

Consider a "simple" switch. In this example we configure the Switch with two commands: to turn the light on and to turn the light off. A benefit of this particular implementation of the command pattern is that the switch can be used with any device, not just a light — the Switch in the following example turns a light on and off, but the Switch's constructor is able to accept any subclasses of Command for its two parameters. For example, you could configure the Switch to start an engine.

Implementation in Java

```
/* The Command interface */
public interface Command {
   void execute();
}
```

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

/* The Invoker class */
public class Switch {
   private ListCommand history = new ArrayListCommand();

   public Switch() {
   }

   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.execute();
   }
}
```

```
/* The 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 for turning on the light - ConcreteCommand #1 */
public class FlipUpCommand implements Command {
   private Light theLight;

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

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

```
/* The Command for turning off the light - ConcreteCommand #2 */
public class FlipDownCommand implements Command {
   private Light theLight;

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

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

```
   }
}


/* The test class or client */
public class PressSwitch {
   public static void main(String[] args){
      Light lamp = new Light();
      Command switchUp = new FlipUpCommand(lamp);
      Command switchDown = new FlipDownCommand(lamp);

      Switch s = new Switch();

      try {
         if (args[0].equalsIgnoreCase("ON")) {
            s.storeAndExecute(switchUp);
         }
         else if (args[0].equalsIgnoreCase("OFF")) {
            s.storeAndExecute(switchDown);
         }
         else {
            System.out.println("Argument \"ON\" or \"OFF\" is required.");
         }
      } catch (Exception e) {
         System.out.println("Arguments required.");
      }
   }
}
```

Implementation in C#

The following code is an implementation of Command pattern in C#.

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

namespace CommandPattern
{
    public interface ICommand
    {
        void Execute();
    }

    /* The Invoker class */
    public class Switch
    {
        private ListICommand _commands = new ListICommand();

        public void StoreAndExecute(ICommand command)
        {
            _commands.Add(command);
            command.Execute();
        }
    }

    /* The Receiver class */
    public class Light
    {
        public void TurnOn()
        {
            Console.WriteLine("The light is on");
        }

        public void TurnOff()
        {
```

```
            Console.WriteLine("The light is off");
        }
    }

    /* The Command for turning on the light - ConcreteCommand #1 */
    public class FlipUpCommand : ICommand
    {
        private Light _light;

        public FlipUpCommand(Light light)
        {
            _light = light;
        }

        public void Execute()
        {
            _light.TurnOn();
        }
    }

    /* The Command for turning off the light - ConcreteCommand #2 */
    public class FlipDownCommand : ICommand
    {
        private Light _light;

        public FlipDownCommand(Light light)
        {
            _light = light;
        }

        public void Execute()
        {
            _light.TurnOff();
        }
    }

    /* The test class or client */
    internal class Program
    {
        public static void Main(string[] args)
        {
            Light lamp = new Light();
            ICommand switchUp = new FlipUpCommand(lamp);
            ICommand switchDown = new FlipDownCommand(lamp);

            Switch s = new Switch();
            string arg = args.Length  0 ? args[0].ToUpper() : null;
            if (arg == "ON")
            {
                s.StoreAndExecute(switchUp);
            }
            else if (arg == "OFF")
            {
                s.StoreAndExecute(switchDown);
            }
            else
            {
                Console.WriteLine("Argument \"ON\" or \"OFF\" is required.");
            }
        }
    }
}
```

Implementation in Python

The following code is an implementation of Command pattern in Python.

```
class Switch(object):
    """The INVOKER class"""
    def __init__(self, flip_up_cmd, flip_down_cmd):
        self.flip_up = flip_up_cmd
        self.flip_down = flip_down_cmd

class Light(object):
    """The RECEIVER class"""
    def turn_on(self):
        print "The light is on"
    def turn_off(self):
        print "The light is off"

class LightSwitch(object):
    """The CLIENT class"""
    def __init__(self):
        lamp = Light()
        self._switch = Switch(lamp.turn_on, lamp.turn_off)

    def switch(self, cmd):
        cmd = cmd.strip().upper()
        if cmd == "ON":
            self._switch.flip_up()
        elif cmd == "OFF":
            self._switch.flip_down()
        else:
            print 'Argument "ON" or "OFF" is required.'

# Execute if this file is run as a script and not imported as a module
if __name__ == "__main__":
    light_switch = LightSwitch()
    print "Switch ON test."
    light_switch.switch("ON")
    print "Switch OFF test."
    light_switch.switch("OFF")
    print "Invalid Command test."
    light_switch.switch("****")
```

Implementation in Scala

```
/* The Command interface */
trait Command {
   def execute()
}

/* The Invoker class */
class Switch {
   private var history: List[Command] = Nil

   def storeAndExecute(cmd: Command) {
      cmd.execute()
      this.history :+= cmd
   }
}

/* The Receiver class */
class Light {
   def turnOn() = println("The light is on")
   def turnOff() = println("The light is off")
}

/* The Command for turning on the light - ConcreteCommand #1 */
class FlipUpCommand(theLight: Light) extends Command {
   def execute() = theLight.turnOn()
```

```
}

/* The Command for turning off the light - ConcreteCommand #2 */
class FlipDownCommand(theLight: Light) extends Command {
   def execute() = theLight.turnOff()
}

/* The test class or client */
object PressSwitch {
   def main(args: Array[String]) {
      val lamp = new Light()
      val switchUp = new FlipUpCommand(lamp)
      val switchDown = new FlipDownCommand(lamp)

      val s = new Switch()

      try {
         args(0).toUpperCase match {
            case "ON" = s.storeAndExecute(switchUp)
            case "OFF" = s.storeAndExecute(switchDown)
            case _ = println("Argument \"ON\" or \"OFF\" is required.")
         }
      } catch {
         case e: Exception = println("Arguments required.")
      }
   }
}
```

Implementation in JavaScript

The following code is an implementation of Command pattern in Javascript.

```
/* The Invoker function */
var Switch = function(){
    this.storeAndExecute = function(command){
        command.execute();
    }
}

/* The Receiver function */
var Light = function(){
    this.turnOn = function(){ console.log ('turn on')};
    this.turnOff = function(){ console.log ('turn off') };
}

/* The Command for turning on the light - ConcreteCommand #1 */
var FlipUpCommand = function(light){
    this.execute = light.turnOn;
}

/* The Command for turning off the light - ConcreteCommand #2 */
var FlipDownCommand = function(light){
    this.execute = light.turnOff;
}

var light = new Light();
var switchUp = new FlipUpCommand(light);
var switchDown = new FlipDownCommand(light);
var s = new Switch();

s.storeAndExecute(switchUp);
s.storeAndExecute(switchDown);
```

Implementation in Smalltalk

```
Object subclass: #Switch
  instanceVariableNames:
    ' flipUpCommand flipDownCommand '
  classVariableNames: ''
  poolDictionaries: ''

Object subclass: #Light
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

Object subclass: #PressSwitch
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''

!Switch class methods !
upMessage: flipUpMessage downMessage: flipDownMessage

^self new upMessage: flipUpMessage downMessage: flipDownMessage; yourself.! !

!Switch methods !
upMessage: flipUpMessage downMessage: flipDownMessage
flipUpCommand := flipUpMessage.
flipDownCommand := flipDownMessage.!

flipDown
flipDownCommand perform.!

flipUp
flipUpCommand perform.! !

!Light methods !
turnOff
Transcript show: 'The light is off'; cr.!

turnOn
Transcript show: 'The light is on'; cr.! !

!PressSwitch class methods !
switch: state
" This is the test method "

| lamp switchUp switchDown switch |
lamp := Light new.
switchUp := Message receiver: lamp selector: #turnOn.
switchDown := Message receiver: lamp selector: #turnOff.

switch := Switch upMessage: switchUp downMessage: switchDown.

state = #on ifTrue: [ ^switch flipUp ].
state = #off ifTrue: [ ^switch flipDown ].

Transcript show: 'Argument #on or #off is required.'.
! !
```

16 Print version

The following Reverse Polish notation[1] example illustrates the interpreter pattern. The grammar: `expression ::= plus | minus | variable | number`

```
plus ::= expression expression '+'

minus ::= expression expression '-'

variable  ::= 'a' | 'b' | 'c' | ... | 'z'

digit = '0' | '1' | ... '9'

number ::= digit | digit number
```

defines a language which contains reverse Polish expressions like:

```
a b +

a b c + -

a b + c a - -
```

Following the interpreter pattern there is a class for each grammar rule.

Implementation in Java

```
import java.util.Map;

interface Expression {
    public int interpret(MapString, Expression variables);
}
```

```
import java.util.Map;

class Number implements Expression {
    private int number;

    public Number(int number) {
        this.number = number;
```

1 http://en.wikibooks.org//en.wikipedia.org/wiki/Reverse_Polish_notation

```
    }

    public int interpret(MapString, Expression variables) {
        return number;
    }
}


import java.util.Map;

class Plus implements Expression {
    Expression leftOperand;

    Expression rightOperand;

    public Plus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(MapString, Expression variables) {
        return leftOperand.interpret(variables) +
 rightOperand.interpret(variables);
    }
}


import java.util.Map;

class Minus implements Expression {
    Expression leftOperand;

    Expression rightOperand;

    public Minus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(MapString, Expression variables)  {
        return leftOperand.interpret(variables) -
 rightOperand.interpret(variables);
    }
}


import java.util.Map;

class Variable implements Expression {
    private String name;

    public Variable(String name) {
        this.name = name;
    }

    public int interpret(MapString, Expression variables) {
        if (variables.get(name) == null) {
            // Either return new Number(0).
            return 0;
        } else {
            return variables.get(name).interpret(variables);
        }
    }
}
```

While the interpreter pattern does not address parsing a parser is provided for completeness.

```
import java.util.Map;
import java.util.Stack;

class Evaluator implements Expression {
    private Expression syntaxTree;

    public Evaluator(String expression) {
        StackExpression expressionStack = new StackExpression();
        for (String token : expression.split(" ")) {
            if  (token.equals("+")) {
                Expression subExpression = new Plus(expressionStack.pop(),
 expressionStack.pop());
                expressionStack.push( subExpression );
            }
            else if (token.equals("-")) {
                // it's necessary remove first the right operand from the stack
                Expression right = expressionStack.pop();
                // ..and after the left one
                Expression left = expressionStack.pop();
                Expression subExpression = new Minus(left, right);
                expressionStack.push( subExpression );
            }
            else
                expressionStack.push( new Variable(token) );
        }
        syntaxTree = expressionStack.pop();
    }

    public int interpret(MapString,Expression context) {
        return syntaxTree.interpret(context);
    }
}
```

Finally evaluating the expression "w x z - +" with w = 5, x = 10, and z = 42.

```
import java.util.Map;
import java.util.HashMap;

public class InterpreterExample {
    public static void main(String[] args) {
        String expression = "w x z - +";
        Evaluator sentence = new Evaluator(expression);
        MapString,Expression variables = new HashMapString,Expression();
        variables.put("w", new Number(5));
        variables.put("x", new Number(10));
        variables.put("z", new Number(42));
        int result = sentence.interpret(variables);
        System.out.println(result);
    }
}
```

Implementation in C#

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

namespace Interpreter
{
    class Program
    {
```

```
        interface IExpression
        {
            int Interpret(Dictionarystring, int variables);
        }

        class Number : IExpression
        {
            public int number;
            public Number(int number) { this.number = number; }
            public int Interpret(Dictionarystring, int variables) { return
number; }
        }

        abstract class BasicOperation : IExpression
        {
            IExpression leftOperator, rightOperator;

            public BasicOperation(IExpression left, IExpression right)
            {
                leftOperator = left;
                rightOperator = right;
            }

            public int Interpret(Dictionarystring, int variables)
            {
                return Execute(leftOperator.Interpret(variables),
rightOperator.Interpret(variables));
            }

            abstract protected int Execute(int left, int right);
        }

        class Plus : BasicOperation
        {
            public Plus(IExpression left, IExpression right) : base(left, right)
{ }

            protected override int Execute(int left, int right)
            {
                return left + right;
            }
        }

        class Minus : BasicOperation
        {
            public Minus(IExpression left, IExpression right) : base(left,
right) { }

            protected override int Execute(int left, int right)
            {
                return left - right;
            }
        }

        class Variable : IExpression
        {
            private string name;

            public Variable(string name) { this.name = name; }

            public int Interpret(Dictionarystring, int variables)
            {
                return variables[name];
            }
        }

        class Evaluator
        {
```

```
            private IExpression syntaxTree;

            public Evaluator(string expression)
            {
                StackIExpression stack = new StackIExpression();
                foreach (string token in expression.Split(' '))
                {
                    if (token.Equals("+"))
                        stack.Push(new Plus(stack.Pop(), stack.Pop()));
                    else if (token.Equals("-")){
                        IExpression right = stack.Pop();
                        IExpression left = stack.Pop();
                        stack.Push(new Minus(left, right));
                    }else
                        stack.Push(new Variable(token));
                }
                syntaxTree = stack.Pop();
            }

            public int Evaluate(Dictionarystring, int context)
            {
                return syntaxTree.Interpret(context);
            }
        }

        static void Main(string[] args)
        {
            Evaluator evaluator = new Evaluator("w x z - +");
            Dictionarystring, int values = new Dictionarystring,int();
            values.Add("w", 5);
            values.Add("x", 10);
            values.Add("z", 42);
            Console.WriteLine(evaluator.Evaluate(values));
        }
    }
}
```

Implementation in PHP

In a file we have the classes and the interface, defining the logic of the program (and applying the Interpreter pattern). Now the *expr.php* file:

```
?php

    interface expression{
        public function interpret (array $variables);
        public function __toString();
    }

    class number implements expression{
        private $number;
        public function __construct($number){
            $this-number = intval($number);
        }
        public function interpret(array $variables){
            return $this-number;
        }
        public function __toString(){
            return (string) $this-number;
        }
    }

    class plus implements expression{
        private $left_op;
        private $right_op;
```

```
        public function __construct(expression $left, expression $right){
            $this-left_op = $left;
            $this-right_op = $right;
        }
        public function interpret(array $variables){
            return ($this-left_op-interpret($variables) +
 $this-right_op-interpret($variables));
        }
        public function __toString(){
            return (string) ("Left op: {$this-left_op} + Right op:
 {$this-right_op}\n");
        }
    }

    class minus implements expression{
        private $left_op;
        private $right_op;
        public function __construct(expression $left, expression $right){
            $this-left_op = $left;
            $this-right_op = $right;
        }
        public function interpret(array $variables){
            return ($this-left_op-interpret($variables) -
 $this-right_op-interpret($variables));
        }
        public function __toString(){
            return (string) ("Left op: {$this-left_op} - Right op:
 {$this-right_op}\n");
        }
    }

    class variable implements expression{
        private $name;
        public function __construct($name){
            $this-name = $name;
        }
        public function interpret(array $variables){
            if(!isset($variables[$this-name]))
                return 0;
            return $variables[$this-name]-interpret($variables);
        }
        public function __toString(){
            return (string) $this-name;
        }
    }

?
```

And the *evaluate.php*:

```
?php

    require_once('expr.php');

    class evaluator implements expression{
        private $syntaxTree;

        public function __construct($expression){
            $stack = array();
            $tokens = explode(" ", $expression);
            foreach($tokens as $token){
                if($token == "+"){
                    $right = array_pop($stack);
                    $left = array_pop($stack);
                    array_push($stack, new plus($left, $right));
                }
```

```
            else if($token == "-"){
                $right = array_pop($stack);
                $left = array_pop($stack);
                array_push($stack, new minus($left, $right));
            }else if(is_numeric($token)){
                array_push($stack, new number($token));
            }else{
                array_push($stack, new variable($token));
            }
        }
        $this-syntaxTree = array_pop($stack);
    }

    public function interpret(array $context){
        return $this-syntaxTree-interpret($context);
    }

    public function __toString(){
        return "";
    }

}

// main code

// works for it:
$expression = "5 10 42 - +";
// or for it:
//$expression = "w x z - +";
$variables = array();
$variables['w'] = new number("5");
$variables['x'] = new number("10");
$variables['z'] = new number("42");
print ("Evaluating expression {$expression}\n");
$sentence = new evaluator($expression);
$result = $sentence-interpret($variables);
print $result . "\n";

?
```

And you can run on a terminal by typing php5 -f evaluator.php (or putting it on a web application).

17 Iterator

Various examples of the iterator pattern.

Implementation in Java

Here is an example in Java[1]:

```
import java.util.BitSet;
import java.util.Iterator;

public class BitSetIterator implements IteratorBoolean {

    private final BitSet bitset;
    private int index = 0;

    public BitSetIterator(BitSet bitset) {
        this.bitset = bitset;
    }

    public boolean hasNext() {
        return index  bitset.length();
    }

    public Boolean next() {
        if (index = bitset.length()) {
            throw new NoSuchElementException();
}
        return bitset.get(index++);
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }

}
```

Two different usage examples:

```
import java.util.BitSet;

public class TestClientBitSet {

    public static void main(String[] args) {
        // create BitSet and set some bits
        BitSet bitset = new BitSet();
        bitset.set(1);
        bitset.set(3400);
        bitset.set(20);
        bitset.set(47);
        for (BitSetIterator iter = new BitSetIterator(bitset); iter.hasNext(); )
 {
```

1 http://en.wikibooks.org/wiki/Subject:Java_programming_language

```
            Boolean b = iter.next();
            String tf = (b.booleanValue() ? "T" : "F");
            System.out.print(tf);
        }
System.out.println();
    }

}
```

```
import java.util.ArrayList;
import java.util.Collection;

public class TestClientIterator  {

    public static void main(String[] args) {
        CollectionObject al = new ArrayListObject();
        al.add(new Integer(42));
        al.add("test");
        al.add(new Double("-12.34"));
        for (IteratorObject iter = al.iterator(); iter.hasNext(); ) {
            System.out.println(iter.next());
        }
        for (Object o : al) {
            System.out.println(o);
        }
    }
}
```

Implementation in C#

Here is an example in C#[2]:

```
using System;
using System.Collections;

  class MainApp
  {
    static void Main()
    {
      ConcreteAggregate a = new ConcreteAggregate();
      a[0] = "Item A";
      a[1] = "Item B";
      a[2] = "Item C";
      a[3] = "Item D";

      // Create Iterator and provide aggregate
      ConcreteIterator i = new ConcreteIterator(a);

      Console.WriteLine("Iterating over collection:");

      object item = i.First();
      while (item != null)
      {
        Console.WriteLine(item);
        item = i.Next();
      }

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

2 http://en.wikibooks.org/wiki/C_Sharp_Programming

```
// "Aggregate"
abstract class Aggregate
{
  public abstract Iterator CreateIterator();
}

// "ConcreteAggregate"
class ConcreteAggregate : Aggregate
{
  private ArrayList items = new ArrayList();

  public override Iterator CreateIterator()
  {
    return new ConcreteIterator(this);
  }

  // Property
  public int Count
  {
    get{ return items.Count; }
  }

  // Indexer
  public object this[int index]
  {
    get{ return items[index]; }
    set{ items.Insert(index, value); }
  }
}

// "Iterator"
abstract class Iterator
{
  public abstract object First();
  public abstract object Next();
  public abstract bool IsDone();
  public abstract object CurrentItem();
}

// "ConcreteIterator"
class ConcreteIterator : Iterator
{
  private ConcreteAggregate aggregate;
  private int current = 0;

  // Constructor
  public ConcreteIterator(ConcreteAggregate aggregate)
  {
    this.aggregate = aggregate;
  }

  public override object First()
  {
    return aggregate[0];
  }

  public override object Next()
  {
    object ret = null;
    if (current  aggregate.Count - 1)
    {
      ret = aggregate[++current];
    }

    return ret;
  }
```

```
    public override object CurrentItem()
    {
      return aggregate[current];
    }

    public override bool IsDone()
    {
      return current = aggregate.Count ? true : false ;
    }
  }
```

Implementation in PHP 5

As a default behavior in PHP[3] 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 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()
while 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.

Implementation in Perl

In Perl[4], objects providing an iterator interface either overload[5] the <> (iterator operator)[1], or provide a hash or tied hash[6] interface that can be iterated over with

3 http://en.wikibooks.org/wiki/PHP
4 http://en.wikibooks.org/wiki/Perl
5 http://perldoc.perl.org/overload.html
6 http://perldoc.perl.org/Tie/Hash.html

`each`[7][2]. Both <> and `each` return `undef` when iteration is complete. Overloaded <> operator:

```
# fibonacci sequence
package FibIter;
use overload
    '' = 'next_fib';
sub new {
    my $class = shift;
    bless { index = 0, values = [0, 1] }, $class
}
sub next_fib {
    my $self = shift;
    my $i = $self-{index}++;
    $self-{values}[$i] ||=
        $i  1 ? $self-{values}[-2]+$self-{values}[-1]
               : ($self-{values}[$i]);
}
# reset iterator index
sub reset {
        my $self = shift;
        $self-{index} = 0
}

package main;
my $iter = FibIter-new;
while (my $fib = $iter) {
    print "$fib","\n";
}
```

Iterating over a hash (or tied hash):

```
# read from a file like a hash
package HashIter;
use base 'Tie::Hash';
sub new {
        my ($class, $fname) = @_;
        my $obj = bless {}, $class;
        tie %$obj, $class, $fname;
        bless $obj, $class;
}

sub TIEHASH {
        # tie hash to a file
        my $class = shift;
        my $fname = shift or die 'Need filename';
        die "File $fname must exist"
                 unless [-f $fname];
        open my $fh, '', $fname or die "open $!";
        bless { fname = $fname, fh = $fh }, $class;
}
sub FIRSTKEY {
        # (re)start iterator
        my $self = shift;
        my $fh = $self-{fh};
        if (not fileno $self-{fh}) {
                open $fh, '', $self-{fname} or die "open $!";
        }
        # reset file pointer
        seek( $fh, 0, 0 );
        chomp(my $line = $fh);
```

7 http://perldoc.perl.org/each.html

```
        $line
}
sub NEXTKEY {
        # next item from iterator
        my $self = shift;
        my $fh = $self-{fh};
        return if eof($fh);
        chomp(my $line = $fh);
        $line
}
sub FETCH {
        # get value for key, in this case we don't
        # care about the values, just return
        my ($self, $key) = @_;
        return
}

sub main;
# iterator over a word file
my $word_iter = HashIter-new('/usr/share/dict/words');
# iterate until we get to abacus
while (my $word = each( %$word_iter )) {
        print "$word\n";
        last if $word eq 'abacus'
}
# call keys %tiedhash in void context to reset iterator
keys %$word_iter;
```

Implementation in Python

In Python[8], iterators are objects that adhere to the *iterator protocol*. You can get an iterator from any sequence (i.e. collection: lists, tuples, dictionaries, sets, etc.) with the `iter()` method. Another way to get an iterator is to create a generator, which is a kind of iterator. To get the next element from an iterator, you use the `next()` method (Python 2) / `next()` function (Python 3). When there are no more elements, it raises the `StopIteration` exception. To implement your own iterator, you just need an object that implements the `next()` method (Python 2) / `__next__()` method (Python 3). Here are two use cases:

```
# from a sequence
x = [42, "test", -12.34]
it = iter(x)
try:
  while True:
    x = next(it) # in Python 2, you would use it.next()
    print x
except StopIteration:
  pass

# a generator
def foo(n):
  for i in range(n):
    yield i

it = foo(5)
try:
  while True:
    x = next(it) # in Python 2, you would use it.next()
    print x
except StopIteration:
  pass
```

8 http://en.wikibooks.org/wiki/Python_Programming

Implementation in MATLAB

MATLAB[9] supports both external and internal implicit iteration using either "native" arrays or `cell` arrays. In the case of external iteration where the onus is on the user to advance the traversal and request next elements, one can define a set of elements within an array storage structure and traverse the elements using the `for`-loop construct. For example,

```
% Define an array of integers
myArray = [1,3,5,7,11,13];

for n = myArray
   % ... do something with n
   disp(n)  % Echo integer to Command Window
end
```

traverses an array of integers using the `for` keyword. In the case of internal iteration where the user can supply an operation to the iterator to perform over every element of a collection, many built-in operators and MATLAB functions are overloaded to execute over every element of an array and return a corresponding output array implicitly. Furthermore, the `arrayfun` and `cellfun` functions can be leveraged for performing custom or user defined operations over "native" arrays and `cell` arrays respectively. For example,

```
function simpleFun
% Define an array of integers
myArray = [1,3,5,7,11,13];

% Perform a custom operation over each element
myNewArray = arrayfun(@(a)myCustomFun(a),myArray);

% Echo resulting array to Command Window
myNewArray


function outScalar = myCustomFun(inScalar)
% Simply multiply by 2
outScalar = 2*inScalar;
```

defines a primary function `simpleFun` which implicitly applies custom subfunction `myCustomFun` to each element of an array using built-in function `arrayfun`. Alternatively, it may be desirable to abstract the mechanisms of the array storage container from the user by defining a custom object-oriented MATLAB implementation of the Iterator Pattern. Such an implementation supporting external iteration is demonstrated in MATLAB Central File Exchange item Design Pattern: Iterator (Behavioural)[10]. This is written in the new class-definition syntax introduced with MATLAB software version 7.6 (R2008a) [3] and features a one-dimensional `cell` array realisation of the List Abstract Data Type (ADT) as the mechanism for storing a heterogeneous (in data type) set of elements. It provides the functionality for explicit forward List traversal with the `hasNext()`, `next()` and `reset()` methods for use in a `while`-loop.

9 `http://en.wikibooks.org/wiki/Matlab`
10 `http://www.mathworks.com.au/matlabcentral/fileexchange/25225`

17.1 References

1. File handle objects implement this to provide line by line reading of their contents
2. In Perl 5.12, arrays and tied arrays[11] can be iterated over like hashes, with `each`
3. "New Class-Definition Syntax Introduced with MATLAB Software Version 7.6"[12]. The MathWorks, Inc. March 2009. [13]. Retrieved September 22, 2009.

11 http://perldoc.perl.org/Tie/Array.html
12 http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_oop/brqzfth-1.html#brqzfth-3
13 http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_oop/brqzfth-1.html#brqzfth-3

18 Print version

This pattern helps to model a class whose object at run-time is responsible for controlling and coordinating the interactions of a group of other objects.

18.1 Participants

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

Implementation in Java

```
// Colleague interface
interface Command {
    void execute();
}
```

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

// Concrete mediator
class Mediator {

    BtnView btnView;
    BtnSearch btnSearch;
    BtnBook btnBook;
    LblDisplay show;

    //....
    void registerView(BtnView v) {
        btnView = v;
    }

    void registerSearch(BtnSearch s) {
        btnSearch = s;
    }

    void registerBook(BtnBook b) {
        btnBook = b;
```

```
    }

    void registerDisplay(LblDisplay d) {
        show = d;
    }

    void book() {
        btnBook.setEnabled(false);
        btnView.setEnabled(true);
        btnSearch.setEnabled(true);
        show.setText("booking...");
    }

    void view() {
        btnView.setEnabled(false);
        btnSearch.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("viewing...");
    }

    void search() {
        btnSearch.setEnabled(false);
        btnView.setEnabled(true);
        btnBook.setEnabled(true);
        show.setText("searching...");
    }

}
```

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
// A concrete colleague
class BtnView extends JButton implements Command {

    Mediator med;

    BtnView(ActionListener al, Mediator m) {
        super("View");
        addActionListener(al);
        med = m;
        med.registerView(this);
    }

    public void execute() {
        med.view();
    }

}
```

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
```

```
// A concrete colleague
class BtnSearch extends JButton implements Command {

    Mediator med;

    BtnSearch(ActionListener al, Mediator m) {
        super("Search");
        addActionListener(al);
        med = m;
        med.registerSearch(this);
    }

    public void execute() {
        med.search();
    }

}
```

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

// A concrete colleague
class BtnBook extends JButton implements Command {

    Mediator med;

    BtnBook(ActionListener al, Mediator m) {
        super("Book");
        addActionListener(al);
        med = m;
        med.registerBook(this);
    }

    public void execute() {
        med.book();
    }

}
```

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

class LblDisplay extends JLabel {

    Mediator med;

    LblDisplay(Mediator m) {
        super("Just start...");
        med = m;
        med.registerDisplay(this);
        setFont(new Font("Arial", Font.BOLD, 24));
    }
```

```
}
```

```
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

class MediatorDemo extends JFrame implements ActionListener {

    Mediator med = new Mediator();

    MediatorDemo() {
        JPanel p = new JPanel();
        p.add(new BtnView(this, med));
        p.add(new BtnBook(this, med));
        p.add(new BtnSearch(this, med));
        getContentPane().add(new LblDisplay(med), "North");
        getContentPane().add(p, "South");
        setSize(400, 200);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent ae) {
        Command comd = (Command) ae.getSource();
        comd.execute();
    }

    public static void main(String[] args) {
        new MediatorDemo();
    }

}
```

Implementation in C#

```
using System;
using System.Collections;

  class MainApp
  {
    static void Main()
    {
      ConcreteMediator m = new ConcreteMediator();

      ConcreteColleague1 c1 = new ConcreteColleague1(m);
      ConcreteColleague2 c2 = new ConcreteColleague2(m);

      m.Colleague1 = c1;
      m.Colleague2 = c2;

      c1.Send("How are you?");
      c2.Send("Fine, thanks");

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

  // "Mediator"
```

```
abstract class Mediator
{
  public abstract void Send(string message,
    Colleague colleague);
}

// "ConcreteMediator"
class ConcreteMediator : Mediator
{
  private ConcreteColleague1 colleague1;
  private ConcreteColleague2 colleague2;

  public ConcreteColleague1 Colleague1
  {
    set{ colleague1 = value; }
  }

  public ConcreteColleague2 Colleague2
  {
    set{ colleague2 = value; }
  }

  public override void Send(string message,
    Colleague colleague)
  {
    if (colleague == colleague1)
    {
      colleague2.Notify(message);
    }
    else
    {
      colleague1.Notify(message);
    }
  }
}

// "Colleague"
abstract class Colleague
{
  protected Mediator mediator;

  // Constructor
  public Colleague(Mediator mediator)
  {
    this.mediator = mediator;
  }
}

// "ConcreteColleague1"
class ConcreteColleague1 : Colleague
{
  // Constructor
  public ConcreteColleague1(Mediator mediator)
    : base(mediator)
  {
  }

  public void Send(string message)
  {
    mediator.Send(message, this);
  }

  public void Notify(string message)
  {
    Console.WriteLine("Colleague1 gets message: "
      + message);
  }
}
```

```
// "ConcreteColleague2"
class ConcreteColleague2 : Colleague
{
  // Constructor
  public ConcreteColleague2(Mediator mediator)
    : base(mediator)
  {
  }

  public void Send(string message)
  {
    mediator.Send(message, this);
  }

  public void Notify(string message)
  {
    Console.WriteLine("Colleague2 gets message: "
      + message);
  }
}
```

19 Memento

Briefly, the Originator (the object to be saved) creates a snap-shot of itself as a Memento object, and passes that reference to the Caretaker object. The Caretaker object keeps the Memento until such a time as the Originator may want to revert to a previous state as recorded in the Memento object.

Implementation in Java

The following Java[1] program illustrates the "undo" usage of the Memento Pattern.

```
class Originator {
    private String state;
    // The class could also contain additional data that is not part of the
    // state saved in the memento.

    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }

    public void restoreFromMemento(Memento memento) {
        state = memento.getSavedState();
        System.out.println("Originator: State after restoring from Memento: " +
 state);
    }

    public static class Memento {
        private final String state;

        private Memento(String stateToSave) {
            state = stateToSave;
        }

        private String getSavedState() {
            return state;
        }
    }
}
```

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

class Caretaker {
    public static void main(String[] args) {
        ListOriginator.Memento savedStates = new ArrayListOriginator.Memento();
```

1 http://en.wikibooks.org/wiki/Subject:Java_programming_language

```
        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to roll back
 to.
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(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
```

Implementation in C#

19.1 C# Example

```
using System;

namespace DoFactory.GangOfFour.Memento.Structural
{
  // MainApp startup class for Structural
  // Memento Design Pattern.
  class MainApp
  {
     // Entry point into console application.
    static void Main()
    {
      Originator o = new Originator();
      o.State = "On";

      // Store internal state
      Caretaker c = new Caretaker();
      c.Memento = o.CreateMemento();

      // Continue changing originator
      o.State = "Off";

      // Restore saved state
      o.SetMemento(c.Memento);

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

  // The 'Originator' class
  class Originator
  {
```

```
    private string _state;

    // Property
    public string State
    {
      get { return _state; }
      set
      {
        _state = value;
        Console.WriteLine("State = " + _state);
      }
    }

    // Creates memento
    public Memento CreateMemento()
    {
      return (new Memento(_state));
    }

    // Restores original state
    public void SetMemento(Memento memento)
    {
      Console.WriteLine("Restoring state...");
      State = memento.State;
    }
  }

  // The 'Memento' class
  class Memento
  {
    private readonly string _state;

    // Constructor
    public Memento(string state) {
      this._state = state;
    }

    // Gets or sets state
    public string State
    {
      get { return _state; }
    }
  }

  // The 'Caretaker' class
  class Caretaker
  {
    private Memento _memento;

    // Gets or sets memento
    public Memento Memento
    {
      set { _memento = value; }
      get { return _memento; }
    }
  }
}
```

19.2 Another way to implement memento in C#

```
public interface IOriginator
{
   IMemento GetState();
```

```
}

public interface IShape : IOriginator
{
   void Draw();
   void Scale(double scale);
   void Move(double dx, double dy);
}

public interface IMemento
{
   void RestoreState();
}

public class CircleOriginator : IShape
{
   private class CircleMemento : IMemento
   {
      private readonly double x;
      private readonly double y;
      private readonly double r;
      private readonly CircleOriginator originator;

      public CircleMemento(CircleOriginator originator)
      {
         this.originator = originator;
         x = originator.x;
         y = originator.y;
         r = originator.r;
      }

      public void RestoreState()
      {
         originator.x = x;
         originator.y = y;
         originator.r = r;
      }
   }

   double x;
   double y;
   double r;

   public CircleOriginator(double x, double y, double r)
   {
      this.x = x;
      this.y = y;
      this.r = r;
   }

   public void Draw()
   {
      Console.WriteLine("Circle with radius {0} at ({1}, {2})", r, x, y);
   }

   public void Scale(double scale)
   {
      r *= scale;
   }

   public void Move(double dx, double dy)
   {
      x += dx;
      y += dy;
   }

   public IMemento GetState()
   {
```

```
      return new CircleMemento(this);
   }
}

public class RectOriginator : IShape
{
   private class RectMemento : IMemento
   {
      private readonly double x;
      private readonly double y;
      private readonly double w;
      private readonly double h;
      private readonly RectOriginator originator;

      public RectMemento(RectOriginator originator)
      {
         this.originator = originator;
         x = originator.x;
         y = originator.y;
         w = originator.w;
         h = originator.h;
      }

      public void RestoreState()
      {
         originator.x = x;
         originator.y = y;
         originator.w = w;
         originator.h = h;
      }
   }

   double x;
   double y;
   double w;
   double h;

   public RectOriginator(double x, double y, double w, double h)
   {
      this.x = x;
      this.y = y;
      this.w = w;
      this.h = h;
   }

   public void Draw()
   {
      Console.WriteLine("Rectangle {0}x{1} at ({2}, {3})", w, h, x, y);
   }

   public void Scale(double scale)
   {
      w *= scale;
      h *= scale;
   }

   public void Move(double dx, double dy)
   {
      x += dx;
      y += dy;
   }

   public IMemento GetState()
   {
      return new RectMemento(this);
   }
}
```

```
public class Caretaker
{
   public void Draw(IEnumerableIShape shapes)
   {
      foreach(IShape shape in shapes)
      {
         shape.Draw();
      }
   }

   public void MoveAndScale(IEnumerableIShape shapes)
   {
      foreach(IShape shape in shapes)
      {
         shape.Scale(10);
         shape.Move(3, 2);
      }
   }

   public IEnumerableIMemento SaveStates(IEnumerableIShape shapes)
   {
      ListIMemento states = new ListIMemento();
      foreach(IShape shape in shapes)
      {
         states.Add(shape.GetState());
      }
      return states;
   }

   public void RestoreStates(IEnumerableIMemento states)
   {
      foreach(IMemento state in states)
      {
         state.RestoreState();
      }
   }

   public static void Main()
   {
      IShape[] shapes = { new RectOriginator(10, 20, 3, 5), new
 CircleOriginator(5, 2, 10) };

      //Outputs:
      // Rectangle 3x5 at (10, 20)
      // Circle with radius 10 at (5, 2)
      Draw(shapes);

      //Save states of figures
      IEnumerableIMemento states = SaveStates(shapes);

      //Change placement of figures
      MoveAndScale(shapes);

      //Outputs:
      // Rectangle 30x50 at (13, 22)
      // Circle with radius 100 at (8, 4)
      Draw(shapes);

      //Restore old placement of figures
      RestoreStates(states);

      //Outputs:
      // Rectangle 3x5 at (10, 20)
      // Circle with radius 10 at (5, 2)
      Draw(shapes);
   }
}
```

Implementation in Pascal

```
{$apptype console}
program Memento;

uses
  SysUtils, Classes;

type
  TMemento = class
  private
    _state: string;
    function GetSavedState: string;
  public
    constructor Create(stateToSave: string);
    property SavedState: string read GetSavedState;
  end;

  TOriginator = class
  private
    _state: string;
    // The class could also contain additional data that is not part of the
    // state saved in the memento.
    procedure SetState(const state: string);
  public
    function SaveToMemento: TMemento;
    procedure RestoreFromMemento(Memento: TMemento);
    property state: string read _state write SetState;
  end;

  TCaretaker = class
  private
    _memento: TMemento;
  public
    property Memento: TMemento read _memento write _memento;
  end;

  { TMemento }

constructor TMemento.Create(stateToSave: string);
begin
  _state := stateToSave;
end;

function TMemento.GetSavedState: string;
begin
  writeln('restoring state from Memento');
  result := _state;
end;

{ TOriginator }

procedure TOriginator.RestoreFromMemento(Memento: TMemento);
begin
  _state := Memento.SavedState;
  writeln('Originator: State after restoring from Memento: ' + state);
end;

function TOriginator.SaveToMemento: TMemento;
begin
  writeln('Originator: Saving to Memento.');
  result := TMemento.Create(state);
end;

procedure TOriginator.SetState(const state: string);
begin
```

```
  writeln('Originator: Setting state to ' + state);
  _state := state;
end;

var
  originator: TOriginator;
  c1,c2: TCaretaker;
begin
  originator := TOriginator.Create;
  originator.SetState('State1');
  originator.SetState('State2');
  // Store internal state
  c1 := TCaretaker.Create();
  c1.Memento := originator.SaveToMemento();
  originator.SetState('State3');
  // We can request multiple mementos, and choose which one to roll back to.
  c2 := TCaretaker.Create();
  c2.Memento := originator.SaveToMemento();
  originator.SetState('State4');

  // Restore saved state
  originator.RestoreFromMemento(c1.Memento);
  originator.RestoreFromMemento(c2.Memento);
  c1.Memento.Free;
  c1.Free;
  c2.Memento.Free;
  c2.Free;
  originator.Free;
end.
```

20 Print version

Define a one-to-many dependency between objects so that when object changes state, all its dependents are notified and update automatically.

Implementation in Python

The observer pattern in Python[1]:

```
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()
```

1 http://en.wikibooks.org/wiki/Python_Programming

```
    # 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[2].

Implementation in Java

Below is an example written in Java[3] that takes keyboard input and treats each input line as an event. The example is built upon the library classes `java.util.Observer`[4] and `java.util.Observable`[5]. When a string is supplied from System.in, the method `notifyObservers` 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.

```
/* Filename : EventSource.java */
package org.wikibooks.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 {
    @Override
    public void run() {
        try {
            final InputStreamReader isr = new InputStreamReader(System.in);
            final BufferedReader br = new BufferedReader(isr);
            while (true) {
                String response = br.readLine();
                setChanged();
                notifyObservers(response);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2 http://en.wikibooks.org/w/index.php?title=Python_syntax_and_semantics&action=edit&redlink=1
3 http://en.wikibooks.org/wiki/Java_Programming
4 http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html
5 http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html

```
/* Filename : ResponseHandler.java */

package org.wikibooks.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 the main program */

package org.wikibooks.obs;

public class MyApp {
    public static void main(String[] args) {
        System.out.println("Enter Text ");

        // create an event source - reads from stdin
        final EventSource eventSource = new EventSource();

        // create an observer
        final ResponseHandler responseHandler = new ResponseHandler();

        // subscribe the observer to the event source
        eventSource.addObserver(responseHandler);

        // starts the event thread
        Thread thread = new Thread(eventSource);
        thread.start();
    }
}
```

Implementation in C#

20.1 Traditional Method

C# and the other .NET Framework[6] languages do not typically require a full implementation of the Observer pattern using interfaces and concrete objects. Here is an example of using them, however.

```
using System;
using System.Collections;

namespace Wikipedia.Patterns.Observer
```

6 http://en.wikibooks.org/wiki/.NET_Framework

```
{
	// 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
			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");
                }
        }
}
```

20.2 Using Events

The alternative to using concrete and abstract observers and publishers in C# and other .NET Framework[7] languages, such as Visual Basic[8], is to use events. The event model is supported via delegates[9] that define the method signature that should be used to capture events. Consequently, delegates provide the mediation otherwise provided by the abstract observer, the methods themselves provide the concrete observer, the concrete subject is the class defining the event, and the subject is the event system built into the base class library. It is the preferred method of accomplishing the Observer pattern in .NET applications.

```
using System;

// First, declare a delegate type that will be used to fire events.
//  This is the same delegate as System.EventHandler.
//  This delegate serves as the abstract observer.
//  It does not provide the implementation, but merely the contract.
public delegate void EventHandler(object sender, EventArgs e);

// Next, declare a published event.  This serves as the concrete subject.
//  Note that the abstract subject is handled implicitly by the runtime.
public class Button
{
    // The EventHandler contract is part of the event declaration.
    public event EventHandler Clicked;

    // By convention, .NET events are fired from descendant classes by a virtual
 method,
    //  allowing descendant classes to handle the event invocation without
 subscribing
    //  to the event itself.
    protected virtual void OnClicked(EventArgs e)
    {
        if (Clicked != null)
            Clicked(this, e); // implicitly calls all observers/subscribers
    }
}

// Then in an observing class, you are able to attach and detach from the
 events:
```

7 http://en.wikibooks.org/wiki/.NET_Framework
8 http://en.wikibooks.org/wiki/Visual_Basic
9 http://en.wikibooks.org/w/index.php?title=Delegation_(programming)&action=edit&redlink=1

```
public class Window
{
    private Button okButton;

    public Window()
    {
        okButton = new Button();
        // This is an attach function.  Detaching is accomplished with -=.
        // Note that it is invalid to use the assignment operator - events are
 multicast
        //  and can have multiple observers.
        okButton.Clicked += new EventHandler(okButton_Clicked);
    }

    private void okButton_Clicked(object sender, EventArgs e)
    {
        // This method is called when Clicked(this, e) is called within the
 Button class
        //  unless it has been detached.
    }
}
```

Implementation in ActionScript 3

```
// Main Class
package {
    import flash.display.MovieClip;

    public class Main extends MovieClip {
        private var _cs:ConcreteSubject = new ConcreteSubject();
        private var _co1:ConcreteObserver1 = new ConcreteObserver1();
        private var _co2:ConcreteObserver2 = new ConcreteObserver2();

        public function Main() {
            _cs.registerObserver(_co1);
            _cs.registerObserver(_co2);

            _cs.changeState(10);
            _cs.changeState(99);

            _cs.unRegisterObserver(_co1);

            _cs.changeState(17);

            _co1 = null;
        }
    }
}

// Interface Subject
package {
    public interface ISubject {
        function registerObserver(o:IObserver):void;

        function unRegisterObserver(o:IObserver):void;

        function updateObservers():void;

        function changeState(newState:uint):void;
    }
}

// Interface Observer
package {
    public interface IObserver {
        function update(newState:uint):void;
```

```
    }
}

// Concrete Subject
package {
    public class ConcreteSubject implements ISubject {
        private var _observersList:Array = new Array();
        private var _currentState:uint;

        public function ConcreteSubject() {
        }

        public function registerObserver(o:IObserver):void {
            _observersList.push( o );
            _observersList[_observersList.length-1].update(_currentState); //
 update newly registered
        }

        public function unRegisterObserver(o:IObserver):void {
            _observersList.splice( _observersList.indexOf( o ), 1 );
        }

        public function updateObservers():void {
            for( var i:uint = 0; i_observersList.length; i++) {
                _observersList[i].update(_currentState);
            }
        }

        public function changeState(newState:uint):void {
            _currentState = newState;
            updateObservers();
        }
    }
}

// Concrete Observer 1
package {
    public class ConcreteObserver1 implements IObserver {
        public function ConcreteObserver1() {
        }

        public function update(newState:uint):void {
            trace( "co1: "+newState );
        }

        // other Observer specific methods
    }
}

// Concrete Observer 2
package {
    public class ConcreteObserver2 implements IObserver {
        public function ConcreteObserver2() {
        }

        public function update(newState:uint):void {
            trace( "co2: "+newState );
        }

        // other Observer specific methods
    }
}
```

Implementation in PHP

class STUDENT

```
?php
class Student implements SplObserver {

        protected $type = "Student";
        private   $name;
        private   $address;
        private   $telephone;
        private   $email;
        private   $_classes = array();

        public function __construct($name)
        {
                $this-name = $name;
        }

        public function GET_type()
        {
                return $this-type;
        }

        public function GET_name()
        {
                return $this-name;
        }

        public function GET_email()
        {
                return $this-email;
        }

        public function GET_telephone()
        {
                return $this-telephone;
        }

        public function update(SplSubject $object)
        {
                $object-SET_log("Comes from ".$this-name.": I'm a student of
 ".$object-GET_materia());
        }

}

?
```

class TEACHER

```
?php
class Teacher implements SplObserver {

        protected $type = "Teacher";
        private   $name;
        private   $address;
        private   $telephone;
        private   $email;
        private   $_classes = array();

        public function __construct($name)
        {
                $this-name = $name;
        }

        public function GET_type()
        {
```

```
                return $this-type;
        }

        public function GET_name()
        {
                return $this-name;
        }

        public function GET_email()
        {
                return $this-email;
        }

        public function GET_telephone()
        {
                return $this-name;
        }

        public function update(SplSubject $object)
        {
                $object-SET_log("Comes from ".$this-name.": I teach in
 ".$object-GET_materia());
        }

}

?
```

Class SUBJECT

```
?php

class Subject implements SplSubject {

        private $name_materia;
        private $_observers = array();
        private $_log = array();

        function __construct($name)
        {
                $this-name_materia = $name;
                $this-_log[]       = "Subject $name was included";
        }

        /* Add an observer */
        public function attach(SplObserver $classes) {
                $this-_classes[] = $classes;
                $this-_log[]     = " The ".$classes-GET_type()." ".$classes-GET_name()."
 was included";
        }

        /* Remove an observer */
        public function detach(SplObserver $classes) {
                foreach ($this-_classes as $key = $obj) {
                        if ($obj == $classes) {
                                unset($this-_classes[$key]);
                                $this-_log[] = " The ".$classes-GET_type()."
 ".$classes-GET_name()." was removed";
                            }
                }
        }

        /* Notificate an observer */
        public function notify(){
                foreach ($this-_classes as $classes){
                        $classes-update($this);
```

```
                }
        }

        public function GET_materia()
        {
                return $this-name_materia;
        }

        function SET_log($valor)
        {
                $this-_log[] = $valor ;
        }

        function GET_log()
        {
                return $this-_log;
        }


}
?
```

Application

```
?php
require_once("teacher.class.php");
require_once("student.class.php");
require_once("subject.class.php");

$subject  = new Subject("Math");
$marcus   = new Teacher("Marcus Brasizza");
$rafael   = new Student("Rafael");
$vinicius = new Student("Vinicius");

// Include observers in the math Subject
$subject-attach($rafael);
$subject-attach($vinicius);
$subject-attach($marcus);

$subject2 = new Subject("English");
$renato   = new Teacher("Renato");
$fabio    = new Student("Fabio");
$tiago    = new Student("Tiago");

// Include observers in the english Subject
$subject2-attach($renato);
$subject2-attach($vinicius);
$subject2-attach($fabio);
$subject2-attach($tiago);

// Remove the instance "Rafael from subject"
$subject-detach($rafael);

// Notify both subjects
$subject-notify();
$subject2-notify();

echo "First Subject br";
echo "pre";
print_r($subject-GET_log());
echo "/pre";
echo "hr";
echo "Second Subject br";
echo "pre";
print_r($subject2-GET_log());
```

```
echo "/pre";
?
```

OUTPUT *First Subject*

```
Array
(
    [0] =  Subject Math was included
    [1] =  The Student Rafael was included
    [2] =  The Student Vinicius was included
    [3] =  The Teacher Marcus Brasizza was included
    [4] =  The Student Rafael was removed
    [5] = Comes from Vinicius: I'm a student of Math
    [6] = Comes from Marcus Brasizza: I teach in Math
)
```

Second Subject

```
Array
(
    [0] =  Subject English was included
    [1] =  The Teacher Renato was included
    [2] =  The Student Vinicius was included
    [3] =  The Student Fabio was included
    [4] =  The Student Tiago was included
    [5] = Comes from Renato: I teach in English
    [6] = Comes from Vinicius: I'm a student of English
    [7] = Comes from Fabio: I'm a student of English
    [8] = Comes from Tiago: I'm a student of English
)
```

Implementation in Ruby

In Ruby, use the standard Observable mixin. For documentation and an example, see [10]

10 `http://www.ruby-doc.org/stdlib/libdoc/observer/rdoc/index.html`

21 Print version

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

Implementation in C#

```
using System;

  class MainApp
  {
    static void Main()
    {
      // Setup context in a state
      Context c = new Context(new ConcreteStateA());

      // Issue requests, which toggles state
      c.Request();
      c.Request();
      c.Request();
      c.Request();

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

  // "State"
  abstract class State
  {
    public abstract void Handle(Context context);
  }

  // "ConcreteStateA"
  class ConcreteStateA : State
  {
    public override void Handle(Context context)
    {
      context.State = new ConcreteStateB();
    }
  }

  // "ConcreteStateB"
  class ConcreteStateB : State
  {
    public override void Handle(Context context)
    {
      context.State = new ConcreteStateA();
    }
  }

  // "Context"
  class Context
  {
    private State state;
```

```
    // Constructor
    public Context(State state)
    {
      this.State = state;
    }

    // Property
    public State State
    {
      get{ return state; }
      set
      {
        state = value;
        Console.WriteLine("State: " +
          state.GetType().Name);
      }
    }

    public void Request()
    {
      state.Handle(this);
    }
  }
```

Implementation in Java

The state interface and two implementations. The state's method has a reference to the context object and is able to change its state.

```java
interface Statelike {

    /**
     * Writer method for the state name.
     * @param STATE_CONTEXT
     * @param NAME
     */
    void writeName(final StateContext STATE_CONTEXT, final String NAME);

}


class StateA implements Statelike {
    /* (non-Javadoc)
     * @see state.Statelike#writeName(state.StateContext, java.lang.String)
     */
    @Override
    public void writeName(final StateContext STATE_CONTEXT, final String NAME) {
        System.out.println(NAME.toLowerCase());
        STATE_CONTEXT.setState(new StateB());
    }

}


class StateB implements Statelike {
    /** State counter */
    private int count = 0;

    /* (non-Javadoc)
     * @see state.Statelike#writeName(state.StateContext, java.lang.String)
     */
    @Override
    public void writeName(final StateContext STATE_CONTEXT, final String NAME) {
        System.out.println(NAME.toUpperCase());
```

```
        // Change state after StateB's writeName() gets invoked twice
        if(++count  1) {
            STATE_CONTEXT.setState(new StateA());
        }
    }

}
```

The context class has a state variable that it instantiates in an initial state, in this case `StateA`. In its method, it uses the corresponding methods of the state object.

```
public class StateContext {
    private Statelike myState;
        /**
         * Standard constructor
         */
    public StateContext() {
        setState(new StateA());
    }

        /**
         * Setter method for the state.
         * Normally only called by classes implementing the State interface.
         * @param NEW_STATE
         */
    public void setState(final Statelike NEW_STATE) {
        myState = NEW_STATE;
    }

        /**
         * Writer method
         * @param NAME
         */
    public void writeName(final String NAME) {
        myState.writeName(this, NAME);
    }
}
```

The test below shows also the usage:

```
public class TestClientState {
    public static void main(String[] args) {
        final StateContext SC = new StateContext();

        SC.writeName("Monday");
        SC.writeName("Tuesday");
        SC.writeName("Wednesday");
        SC.writeName("Thursday");
        SC.writeName("Friday");
        SC.writeName("Saturday");
        SC.writeName("Sunday");
    }
}
```

According to the above code, the output of main() from TestClientState should be:

```
monday
TUESDAY
WEDNESDAY
thursday
FRIDAY
SATURDAY
```

sunday

Implementation in Perl

```
use strict;
use warnings;

package State;
# base state, shared functionality
use base qw{Class::Accessor};
# scan the dial to the next station
sub scan {
	my $self = shift;
	printf "Scanning... Station is %s %s\n",
		$self-{stations}[$self-{pos}], $self-{name};
	$self-{pos}++;
	if ($self-{pos} = @{$self-{stations}}) {
		$self-{pos} = 0
	}
}

package AmState;
our @ISA = qw(State);
AmState-mk_accessors(qw(radio pos name));
use Scalar::Util 'weaken';
sub new {
	my ($class, $radio) = @_;
	my $self;
	@$self{qw(stations pos name radio)} =
		([1250,1380,1510], 0, 'AM', $radio);
	# make circular reference weak
	weaken $self-{radio};
	bless $self, $class;
}
sub toggle_amfm {
	my $self = shift;
	print "Switching to FM\n";
	$self-radio-state( $self-radio-fmstate );
}

package FmState;
our @ISA = qw(State);
FmState-mk_accessors(qw(radio pos name));
use Scalar::Util 'weaken';
sub new {
	my ($class, $radio) = @_;
	my $self;
	@$self{qw(stations pos name radio)} =
		([81.3,89.3,103.9], 0, 'FM', $radio);
	# make circular reference weak
	weaken $self-{radio};
	bless $self, $class;
}
sub toggle_amfm {
	my $self = shift;
	print "Switching to AM\n";
	$self-radio-state( $self-radio-amstate );
}

package Radio;
# this is a radio, it has a scan button and a am/fm toggle
use base qw{Class::Accessor};
Radio-mk_accessors(qw(amstate fmstate state));
sub new {
	my $class = shift;
```

```
        my $self  = {};
        bless $self, $class;

        @$self{ 'amstate', 'fmstate' }
                = ( AmState-new($self), FmState-new($self), );
        $self-state( $self-amstate );
        $self;
}
sub toggle_amfm {
        shift-state-toggle_amfm;
}
sub scan {
        shift-state-scan;
}

package main;
# test out our radio
sub main {
        my $radio = Radio-new;
        my @actions = (
                ('scan')x2,
                ('toggle_amfm'),
                ('scan')x2
        )x2;

        for my $action (@actions) {
                $radio-$action;
        }
        exit;

}

main();
```

Implementation in Ruby

In the Ruby example below, a radio can switch to two states AM and FM and has a scan button to switch to the next station.

```
class State
    def scan
        @pos += 1
        @pos = 0 if @pos == @stations.size
        puts "Scanning… Station is", @stations[@pos], @name
    end
end

class AmState  State
    attr_accessor :radio, :pos, :name
    def initialize radio
        @radio = radio
        @stations = ["1250", "1380", "1510"]
        @pos = 0
        @name = "AM"
    end

    def toggle_amfm
        puts "Switching to FM"
        @radio.state = @radio.fmstate
    end
end

class FmState  State
    attr_accessor :radio, :pos, :name
    def initialize radio
```

```
        @radio = radio
        @stations = ["81.3", "89.1", "103.9"]
        @pos = 0
        @name = "FM"
    end

    def toggle_amfm
        puts "Switching to AM"
        @radio.state = @radio.amstate
    end
end

class Radio
    attr_accessor :amstate, :fmstate, :state
    def initialize
        @amstate = AmState.new self
        @fmstate = FmState.new self
        @state = @amstate
    end

    def toggle_amfm
        @state.toggle_amfm
    end

    def scan
        @state.scan
    end
end

# Test Radio
radio = Radio.new
radio.scan
radio.toggle_amfm # Toggle the state
radio.scan
```

Implementation in Python

```
import itertools

"""Implementation of the state pattern"""
class State(object):
    """Base state. This is to share functionality"""

    def scan(self):
        """Scan the dial to the next station"""
        print "Scanning... Station is", self.stations.next(), self.name

class AmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = itertools.cycle(["1250", "1380", "1510"])
        self.name = "AM"

    def toggle_amfm(self):
        print "Switching to FM"
        self.radio.state = self.radio.fmstate

class FmState(State):
    def __init__(self, radio):
        self.radio = radio
        self.stations = itertools.cycle(["81.3", "89.1", "103.9"])
        self.name = "FM"

    def toggle_amfm(self):
        print "Switching to AM"
        self.radio.state = self.radio.amstate
```

```
class Radio(object):
    """A radio.
    It has a scan button, and an AM/FM toggle switch."""

    def __init__(self):
        """We have an AM state and an FM state"""

        self.amstate = AmState(self)
        self.fmstate = FmState(self)
        self.state = self.amstate

    def toggle_amfm(self):
        self.state.toggle_amfm()

    def scan(self):
        self.state.scan()

def main():
    ''' Test our radio out '''
    radio = Radio()
    actions = ([radio.scan] * 2 + [radio.toggle_amfm] + [radio.scan] * 2) * 2
    for action in actions:
        action()

if __name__ == '__main__':
    main()
```

[1] According to the above Perl and Python code, the output of main() should be:

```
Scanning... Station is 1250 AM
Scanning... Station is 1380 AM
Switching to FM
Scanning... Station is 81.3 FM
Scanning... Station is 89.1 FM
Scanning... Station is 103.9 FM
Scanning... Station is 81.3 FM
Switching to AM
Scanning... Station is 1510 AM
Scanning... Station is 1250 AM
```

21.1 References

1. File handle objects implement this to provide line by line reading of their contents
2. In Perl 5.12, arrays and tied arrays[1] can be iterated over like hashes, with `each`
3. "New Class-Definition Syntax Introduced with MATLAB Software Version 7.6"[2]. The MathWorks, Inc. March 2009. [3]. Retrieved September 22, 2009.

1 `http://perldoc.perl.org/Tie/Array.html`

2 `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_oop/brqzfth-1.html#brqzfth-3`

3 `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_oop/brqzfth-1.html#brqzfth-3`

22 Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Implementation in Ruby

An example in Ruby[1]:

```
class Context
  def initialize(strategy)
    extend(strategy)
  end
end

module StrategyA
  def execute
     puts 'Doing the task the normal way'
  end
end

module StrategyB
  def execute
     puts 'Doing the task alternatively'
  end
end

module StrategyC
  def execute
     puts 'Doing the task even more alternatively'
  end
end

a = Context.new(StrategyA)
a.execute #= Doing the task the normal way

b = Context.new(StrategyB)
b.execute #= Doing the task alternatively

a.execute #= Doing the task the normal way

c = Context.new(StrategyC)
c.execute #= Doing the task even more alternatively
```

22.0.1 Using blocks

The previous ruby example uses typical OO features, but the same effect can be accomplished with ruby's blocks in much less code.

1 http://en.wikibooks.org/wiki/Ruby_Programming

```
class Context
  def initialize(strategy)
    @strategy = strategy
  end

  def execute
    @strategy.call
  end
end

a = Context.new { puts 'Doing the task the normal way' }
a.execute #= Doing the task the normal way

b = Context.new { puts 'Doing the task alternatively' }
b.execute #= Doing the task alternatively

c = Context.new { puts 'Doing the task even more alternatively' }
c.execute #= Doing the task even more alternatively
```

Implementation in Common Lisp

An example in Common Lisp[2]: Using strategy classes:

```
(defclass context ()
  ((strategy :initarg :strategy :accessor strategy)))

(defmethod execute ((c context))
  (execute (slot-value c 'strategy)))

(defclass strategy-a () ())

(defmethod execute ((s strategy-a))
  (print "Doing the task the normal way"))

(defclass strategy-b () ())

(defmethod execute ((s strategy-b))
  (print "Doing the task alternatively"))

(execute (make-instance 'context
                        :strategy (make-instance 'strategy-a)))
```

In Common Lisp using first class functions:

```
(defclass context ()
  ((strategy :initarg :strategy :accessor strategy)))

(defmethod execute ((c context))
  (funcall (slot-value c 'strategy)))

(let ((a (make-instance 'context
                        :strategy (lambda ()
                                    (print "Doing the task the normal way")))))
  (execute a))

(let ((b (make-instance 'context
                        :strategy (lambda ()
                                    (print "Doing the task alternatively")))))
  (execute b))
```

2 http://en.wikibooks.org/wiki/Common_Lisp

Implementation in Java

An example in Java[3]:

```
/** The classes that implement a concrete strategy should implement this.
* The Context class uses this to call the concrete strategy. */
interface Strategy {
    int execute(int a, int b);
}



/** Implements the algorithm using the strategy interface */
class Add implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Add's execute()");
        return a + b;  // Do an addition with a and b
    }
}



class Subtract implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Subtract's execute()");
        return a - b;  // Do a subtraction with a and b
    }
}



class Multiply implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Multiply's execute()");
        return a * b;   // Do a multiplication with a and b
    }
}



/** Configured with a ConcreteStrategy object and maintains a reference to a
 Strategy object */
class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return this.strategy.execute(a, b);
    }
}



/** Tests the pattern */
class StrategyExample {
    public static void main(String[] args) {
        Context context;

        // Three contexts following different strategies
        context = new Context(new Add());
        int resultA = context.executeStrategy(3, 4);
```

3 http://en.wikibooks.org/wiki/Java_Programming

```
        context = new Context(new Subtract());
        int resultB = context.executeStrategy(3, 4);

        context = new Context(new Multiply());
        int resultC = context.executeStrategy(3, 4);

        System.out.println("Result A : " + resultA );
        System.out.println("Result B : " + resultB );
        System.out.println("Result C : " + resultC );
    }
}
```

Implementation in Groovy

This Groovy example is a basic port of the Ruby using blocks example. In place of Ruby's blocks, the example uses Groovy's closure support.

```
class Context {
  def strategy

  Context(strategy) {
    this.strategy = strategy
  }

  def execute() {
    strategy()
  }
}

def a = new Context({ println 'Style A' })
a.execute() // = Style A
def b = new Context({ println 'Style B' })
b.execute() // = Style B
def c = new Context({ println 'Style C' })
c.execute() // = Style C
```

Implementation in Python

An example in Python[4]:

```
class Strategy:
    def execute(a, b):
        pass


class Add(Strategy):
    def execute(self, a, b):
        return a + b


class Subtract(Strategy):
    def execute(self, a, b):
        return a - b


class Multiply(Strategy):
    def execute(self, a, b):
        return a * b
```

4 http://en.wikibooks.org/wiki/Python

```python
class Context:
    def __init__(self, strategy):
        self.strategy = strategy

    def execute(self, a, b):
        return self.strategy.execute(a, b)


if __name__ == "__main__":
    context = None

    context = Context(Add())
    print "Add Strategy %d" % context.execute(10, 5)

    context = Context(Subtract())
    print "Subtract Strategy %d" % context.execute(10, 5)

    context = Context(Multiply())
    print "Multiply Strategy %d" % context.execute(10, 5)
```

Another example in Python: Python has first-class functions, so the pattern can be used simply by passing the function directly to the context instead of defining a class with a method containing the function. One loses information because the interface of the strategy is not made explicit, however, by simplifying the pattern in this manner. Here's an example you might encounter in GUI programming, using a callback function:

```python
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"
```

Implementation in Scala

Like Python, Scala[5] also supports first-class functions. The following implements the basic functionality shown in the Python example.

```scala
  // A very basic button widget.
  class Button[T](val label: String, val onSubmit: Range = T)

  val button1 = new Button("Add", _ reduceLeft (_ + _))
  val button2 = new Button("Join", _ mkString " ")

  // Test each button
  val numbers = 1 to 9  // A list of numbers 1 through 9
  println(button1 onSubmit numbers) // displays 45
  println(button2 onSubmit numbers) // displays 1 2 3 4 5 6 7 8 9
```

Implementation in Falcon

5 http://en.wikibooks.org/wiki/Scala

Similar to Python and Scala, Falcon[6] supports first-class functions. The following implements the basic functionality seen in the Python example.

```
/*#
  @brief A very basic button widget
*/
class Button( label, submit_func )
  label = label
  on_submit = submit_func
end

// Create two instances with different strategies ...
// ... and different ways to express inline functions
button1 = Button( "Add 'em",
    function(nums)
       n = 0
       for val in nums: n+= val
       return n
     end )
button2 = Button( "Join 'em", { nums = " ".merge( [].comp(nums) ) } )

// Test each button
numbers = [1: 10]
printl(button1.on_submit(numbers))   // displays "45"
printl(button2.on_submit(numbers))   // displays "1 2 3 4 5 6 7 8 9"
```

Implementation in JavaScript

Similar to Python and Scala, JavaScript[7] supports first-class functions. The following implements the basic functionality seen in the Python example.

```
var Button = function(submit_func, label) {
        this.label = label;
        this.on_submit = submit_func;
};

var numbers = [1,2,3,4,5,6,7,8,9];
var sum = function(n) {
        var sum = 0;
        for ( var a in n ) {
                sum = sum + n[a];
        }
        return sum;
};

var a = new Button(sum, "Add numbers");
var b = new Button(function(numbers) {
        return numbers.join(',');
}, "Print numbers");

a.on_submit(numbers);
b.on_submit(numbers);
```

Implementation in C

6 http://en.wikibooks.org/w/index.php?title=Falcon_Programming&action=edit&redlink=1
7 http://en.wikibooks.org/wiki/JavaScript

A struct in C[8] can be used to define a class, and the strategy can be set using a function pointer. The following mirrors the Python example, and uses C99 features:

```
#include stdio.h

void print_sum(int n, int *array) {
    int total = 0;
    for (int i = 0; i  n; i++)
        total += array[i];
    printf("%d", total);
}

void print_array(int n, int *array) {
    for (int i = 0; i  n; i++)
        printf("%d ", array[i]);
}

typedef struct {
    void (*submit_func)(int n, int *array);  // function pointer
    char *label;                             // instance label
} Button;

int main(void) {
    // Create two instances with different strategies
    Button button1 = { print_sum, "Add 'em" };
    Button button2 = { print_array, "List 'em" };

    int n = 10;
    int numbers[n];
    for (int i = 0; i  n; i++)
        numbers[i] = i;

    button1.submit_func(n, numbers);
    button2.submit_func(n, numbers);

    return 0;
}
```

Implementation in C++

The strategy pattern in C++[9] is similar to Java, but does not require dynamic allocation of objects.

```
#include iostream

class Strategy
{
public:
     virtual int execute (int a, int b) = 0; // execute() is a so-called pure
 virtual function
                                             // as a consequence, Strategy is a
 so-called abstract class
};

class ConcreteStrategyAdd:public Strategy
{
public:
    int execute(int a, int b)
    {
```

8 http://en.wikibooks.org/wiki/C_Programming
9 http://en.wikibooks.org/wiki/C%2B%2B_Programming

```
        std::cout  "Called ConcreteStrategyAdd's execute()\n";
        return a + b;
    }
};

class ConcreteStrategySubstract:public Strategy
{
public:
    int execute(int a, int b)
    {
        std::cout  "Called ConcreteStrategySubstract's execute()\n";
        return a - b;
    }
};

class ConcreteStrategyMultiply:public Strategy
{
public:
    int execute(int a, int b)
    {
        std::cout  "Called ConcreteStrategyMultiply's execute()\n";
        return a * b;
    }
};

class Context
{
private:
    Strategy* pStrategy;

public:

    Context (Strategy strategy)
        : pStrategy(strategy)
    {
    }

    void SetStrategy(Strategy strategy)
    {
        pStrategy = strategy;
    }

    int executeStrategy(int a, int b)
    {
        return pStrategy-execute(a,b);
    }
};

int main()
{
    ConcreteStrategyAdd       concreteStrategyAdd;
    ConcreteStrategySubstract concreteStrategySubstract;
    ConcreteStrategyMultiply  concreteStrategyMultiply;

    Context context(concreteStrategyAdd);
    int resultA = context.executeStrategy(3,4);

    context.SetStrategy(concreteStrategySubstract);
    int resultB = context.executeStrategy(3,4);

    context.SetStrategy(concreteStrategyMultiply);
    int resultC = context.executeStrategy(3,4);

    std::cout  "resultA: "  resultA  "\tresultB: "  resultB  "\tresultC: "
 resultC  "\n";
}
```

Implementation in C#

Delegates in C#[10] follow the strategy pattern, where the delegate definition defines the strategy interface and the delegate instance represents the concrete strategy. .NET 3.5 defines the Func<,> delegate which can be used to quickly implement the strategy pattern as shown in the example below. Note the 3 different methods for defining a delegate instance.

```
using System;
using System.Linq;
class Program
{
    static void Main(string[] args)
    {
        var context = new Contextint();

        // Delegate
        Funcint, int, int concreteStrategy1 = PerformLogicalBitwiseOr;

        // Anonymous Delegate
        Funcint, int, int concreteStrategy2 = delegate(int op1, int op2) {
 return op1  op2; };

        // Lambda Expressions
        Funcint, int, int concreteStrategy3 = (op1, op2) = op1  op2;
        Funcint, int, int concreteStrategy4 = (op1, op2) = op1  op2;

        context.Strategy = concreteStrategy1;
        var result1 = context.Execute(8, 9);
        context.Strategy = concreteStrategy2;
        var result2 = context.Execute(8, 9);
        context.Strategy = concreteStrategy3;
        var result3 = context.Execute(8, 1);
        context.Strategy = concreteStrategy4;
        var result4 = context.Execute(8, 1);
    }

    static int PerformLogicalBitwiseOr(int op1, int op2)
    {
        return op1 | op2;
    }

    class ContextT
    {
        public FuncT, T, T Strategy { get; set; }

        public T Execute(T operand1, T operand2)
        {
            return this.Strategy != null
                ? this.Strategy(operand1, operand2)
                : default(T);
        }
    }
}
```

22.0.2 Using interfaces

```
using System;

namespace Wikipedia.Patterns.Strategy
```

10 http://en.wikibooks.org/wiki/C_Sharp_Programming

```
{
  // The strategy we will implement will be
  // to advise on investments.
  interface IHasInvestmentStrategy
  {
    long CalculateInvestment();
  }
  // Here we have one way to go about it.
  class FollowTheMoon : IHasInvestmentStrategy
  {
    protected virtual int MoonPhase { get; set; }
    protected virtual int AstrologicalSign { get; set; }
    public FollowTheMoon(int moonPhase, int yourSign)
    {
      MoonPhase = moonPhase;
      AstrologicalSign = yourSign;
    }
    public long CalculateInvestment()
    {
      if (MoonPhase == AstrologicalSign)
        return 1000;
      else
        return 100 * (MoonPhase % DateTime.Today.Day);
    }
  }
  // And here we have another.
  // Note that each strategy may have its own dependencies.
  // The EverythingYouOwn strategy needs a bank account.
  class EverythingYouOwn : IHasInvestmentStrategy
  {
    protected virtual OtherLib.IBankAccessor Accounts { get; set; }
    public EverythingYouOwn(OtherLib.IBankAccessor accounts)
    {
      Accounts = accounts;
    }
    public long CalculateInvestment()
    {
      return Accounts.GetAccountBalancesTotal();
    }
  }
  // The InvestmentManager is where we want to be able to
  // change strategies. This is the Context.
  class InvestmentManager
  {
    public IHasInvestmentStrategy Strategy { get; set; }
    public InvestmentManager(IHasInvestmentStrategy strategy)
    {
      Strategy = strategy;
    }
    public void Report()
    {
      // Our investment is determined by the current strategy.
      var investment = Strategy.CalculateInvestment();
      Console.WriteLine("You should invest {0} dollars!",
        investment);
    }
  }
  class Program
  {
    static void Main()
    {
      // Define some of the strategies we will use.
      var strategyA = new FollowTheMoon( 8, 8 );
      var strategyB = new EverythingYouOwn(
        OtherLib.BankAccountManager.MyAccount);
      // Our investment manager
      var manager = new InvestmentManager(strategyA);
      manager.Report();
```

```
      // You should invest 1000 dollars!
      manager.Strategy = strategyB;
      manager.Report();
      // You should invest 13521500000000 dollars!
    }
  }
}
```

Implementation in ActionScript 3

The strategy pattern in ActionScript[11] 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
        {
```

11 http://en.wikibooks.org/wiki/ActionScript

```
            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();
        }
    }
}
```

Implementation in PHP

The strategy pattern in PHP[12]:

```
?php
interface IStrategy {
    public function execute();
}

class Context {
    private $strategy;

    public function __construct(IStrategy $strategy) {
        $this-strategy = $strategy;
    }

    public function execute() {
        $this-strategy-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 StrategyExample {
```

12 http://en.wikibooks.org/wiki/PHP

```
    public function __construct() {
        $context = new Context(new ConcreteStrategyA());
        $context-execute();

        $context = new Context(new ConcreteStrategyB());
        $context-execute();

        $context = new Context(new ConcreteStrategyC());
        $context-execute();
    }
}

new StrategyExample();
?
```

Implementation in Perl

Perl[13] has first-class functions, so as with Python, JavaScript and Scala, this pattern can be implemented without defining explicit subclasses and interfaces:

```
sort { lc($a) cmp lc($b) } @items
```

The strategy pattern can be formally implemented with Moose[14]:

```
package Strategy;
use Moose::Role;
requires 'execute';


package FirstStrategy;
use Moose;
with 'Strategy';

sub execute {
    print "Called FirstStrategy-execute()\n";
}


package SecondStrategy;
use Moose;
with 'Strategy';

sub execute {
    print "Called SecondStrategy-execute()\n";
}


package ThirdStrategy;
use Moose;
with 'Strategy';

sub execute {
    print "Called ThirdStrategy-execute()\n";
}


package Context;
use Moose;
```

13 http://en.wikibooks.org/wiki/Perl
14 http://en.wikibooks.org/w/index.php?title=Moose_(Perl)&action=edit&redlink=1

```
has 'strategy' = (
    is = 'rw',
    does = 'Strategy',
    handles = [ 'execute' ],  # automatic delegation
);


package StrategyExample;
use Moose;

# Moose's constructor
sub BUILD {
    my $context;

    $context = Context-new(strategy = 'FirstStrategy');
    $context-execute;

    $context = Context-new(strategy = 'SecondStrategy');
    $context-execute;

    $context = Context-new(strategy = 'ThirdStrategy');
    $context-execute;
}


package main;

StrategyExample-new;
```

Implementation in Fortran

Fortran[15] 2003 adds procedure pointers, abstract interfaces and also first-class functions. The following mirrors the Python example.

```
module m_strategy_pattern
implicit none

abstract interface
    !! A generic interface to a subroutine accepting array of integers
    subroutine generic_function(numbers)
        integer, dimension(:), intent(in) :: numbers
    end subroutine
end interface

type :: Button
    character(len=20) :: label
    procedure(generic_function), pointer, nopass :: on_submit
contains
    procedure :: init
end type Button

contains

    subroutine init(self, func, label)
        class(Button), intent(inout) :: self
        procedure(generic_function) :: func
        character(len=*) :: label
        self%on_submit = func      !! Procedure pointer
        self%label = label
    end subroutine init

    subroutine summation(array)
```

15 http://en.wikibooks.org/wiki/Fortran

```
        integer, dimension(:), intent(in) :: array
        integer :: total
        total = sum(array)
        write(*,*) total
    end subroutine summation

    subroutine join(array)
        integer, dimension(:), intent(in) :: array
        write(*,*) array        !! Just write out the whole array
    end subroutine join

end module m_strategy_pattern

!! The following program demonstrates the usage of the module
program test_strategy
use m_strategy_pattern
implicit none

    type(Button) :: button1, button2
    integer :: i

    call button1%init(summation, "Add them")
    call button2%init(join, "Join them")

    call button1%on_submit([(i, i=1,10)])   !! Displays 55
    call button2%on_submit([(i, i=1,10)])   !! Prints out the array

end program test_strategy
```

Implementation in PowerShell

PowerShell[16] has first-class functions called ScriptBlocks, so the pattern can be modeled like Python, passing the function directly to the context instead of defining a class.

```
Function Context ([scriptblock]$script:strategy){
    New-Module -Name Context -AsCustomObject {
        Function Execute {  $strategy }
    }
}

$a = Context {'Style A'}
$a.Execute()

(Context {'Style B'}).Execute()

$c = Context {'Style C'}
$c.Execute()
```

An alternative to using New-Module

```
Function Context ([scriptblock]$strategy){
    {  $strategy }.GetNewClosure()
}

$a = Context {'Style A'}
$a.Invoke()

 (Context {'Style B'})
```

16 http://en.wikibooks.org/wiki/Introduction_to_.NET_Framework_3.0/Windows_Powershell

```
$c = Context {'Style C'}
 $c
```

23 Print version

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of algorithm without changing the algorithm's structure.

Implementation 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 {

    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    /* A template method : */
    public 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() {
        // Initialize players
        // Initialize money
    }

    void makePlay(int player) {
        // Process one turn of player
    }

    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
```

```
    void printWinner() {
        // Display who won
    }

    /* Specific declarations for the Monopoly game. */

    // ...
}
```

```
import java.util.Random;

class SnakesAndLadders extends Game {

    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        playerPositions = new int[playersCount];
        for (int i = 0; i  playersCount; i++) {
           playerPositions[i] = 0;
        }

        die = new Random();

        winnerId = -1;
    }

    void makePlay(int player) {
        // Roll the die
        int  dieRoll = die.nextInt(6) + 1;

        // Move the token
        playerPositions[player] += dieRoll;

        // Move up or down because of the ladders or the snakes
        int penaltyOrBonus = board[playerPositions[player]];
        playerPositions[player] += penaltyOrBonus;

        if (playerPositions[player]  8) {
           // Has reached the top square
           winnerId = player;
        }
    }

    boolean endOfGame() {
        // The game is over when a winner exists
        return (winnerId != -1);
    }

    void printWinner() {
        System.out.println("Player #" + winnerId + " has won!");
    }

    /* Specific declarations for the Snakes and Ladders game. */

    // The board from the bottom square to the top square
    // Each integer is a square
    // Negative values are snake heads with their lengths
    // Positive values are ladder bottoms with their heights
    private static final int[] board = {0, 0, -1, 0, 3, 0, 0, 0, -5, 0};

    // The player positions
    // Each integer represents one player
    // The integer is the position of the player (index) on the board
    private int[] playerPositions = null;

    private Random die = null;
```

```
    private int winnerId = -1;
}
```

Implementation in C#

```
using System;

  class MainApp
  {
    static void Main()
    {
      AbstractClass c;

      c = new ConcreteClassA();
      c.TemplateMethod();

      c = new ConcreteClassB();
      c.TemplateMethod();

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

  // "AbstractClass"
  abstract class AbstractClass
  {
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();
    // The "Template method"
    public void TemplateMethod()
    {
      PrimitiveOperation1();
      PrimitiveOperation2();
      Console.WriteLine("");
    }
  }

  // "ConcreteClass"
  class ConcreteClassA : AbstractClass
  {
    public override void PrimitiveOperation1()
    {
      Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
      Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
    }
  }

  class ConcreteClassB : AbstractClass
  {
    public override void PrimitiveOperation1()
    {
      Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
      Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");
    }
  }
```

24 Print version

The visitor pattern is something like a variation on the Iterator pattern[1] except it allows even more encapsulation. The purpose is to provide a way of performing a certain operation (function, algorithm, etc) for each element in some sort of collection of elements. The basic premise is that you have a `Visitor` interface with a method named `visit` which take one argument, and the collection has a method (typically named `foreach`) which takes a `Visitor` as the argument. This method of the collection will use some encapsulated way of stepping through each of its elements, and for each element, it will invoke the `Visitors`'s `visit` method, passing as the argument the current element. This process is essentially equivalent to getting the collection's iterator[2], and using a `while(iterator.hasNext())` loop to invoke `visitor.visit(iterator.next())`. The key difference is that the collection can decide exactly how to step through the elements. If you're familiar with iterators, you may be thinking that the iterator can decide how to step through, too, and the iterator is usually defined by the collection, so what's the difference? The difference is that the iterator is still bound to a *while* loop, visiting each element in succession, one at a time. With the visitor pattern, the collection could conceivably create a separate thread for each element and have the visitor visiting them concurrently. That's just one example of ways that the collection may decide to vary the method of visitation in a manner that can't be mimicked by an iterator. The point is, it's encapsulated, and the collection can implement the pattern in what ever way it feels is best. More importantly, it can change the implementation at any time, without affecting client code, because the implementation is encapsulated in the `foreach` method.

24.1 An Example implementation of Visitor Pattern: String

To illustrate a simple implementation of visitor pattern, let's imagine we're reinventing Java's String class (it'll be a pretty ridiculous reinvention, but it'll be good for this exercise). We're not going to implement very much of the class, but let's assume that we're storing a set of `char`s that make up the string, and we have a method called `getCharAt` which takes an `int` as it's only argument, and returns the character at that position in the string, as a `char`. We also have a method called `length` which takes no arguments, and returns an `int` which gives the number of characters in the string. Let's also assume that we want to provide an implementation of the visitor pattern for this class which will take an instance that implements the `CharacterVisitor` interface (which we'll define, below), and calls its `visit` method for each character in the string. First we need to define what this `CharacterVisitor` interface looks like:

1 http://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Iterator
2 http://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Iterator

```
public interface CharacterVisitor {

  public void visit(char aChar);

}
```

Easy enough. Now let's get down to our class, which we'll call `MyString`, and it looks something like this:

```
public class MyString
{

// ... other methods, fields

  // Our main implementation of the visitor pattern
  public void foreach(CharacterVisitor aVisitor) {
    int length = this.length();
    // Loop over all the characters in the string
    for (int i = 0; i  length; i++) {
      // Get the current character, and let the visitor visit it.
      aVisitor.visit(this.getCharAt(i));
    }
  }

// ... other methods, fields

}// end class MyString
```

So that was pretty painless. So what can we do with this? Well let's make a class called `MyStringPrinter`, which prints an instance of `MyString` to the standard output.

```
class MyStringPrinter implements CharacterVisitor {

  // We have to implement this method because we're implementing the
 CharacterVisitor
  // interface
  public void visit(char aChar){
    // All we're going to do is print the current character to the standard
 output
    System.out.print(aChar);
  }

  // This is the method you call when you want to print a string
  public void print(MyString aStr){
    // we'll let the string determine how to get each character, and
    // we already defined what to do with each character in our
    // visit method.
    aStr.foreach(this);
  }

} // end class MyStringPrinter
```

That was simple too. Of course, it could've been a lot simpler, right? We didn't need the `foreach` method in `MyString`, and we didn't need `MyStringPrinter` to implement the visitor, we could have just used the `MyString` classes `getCharAt` and `length` methods to set up our own `for` loop ourselves and printed each char inside the loop. Well sure you could, but what if `MyString` isn't `MyString` but instead is `MyBoxOfRocks`. In a box of rocks, is there a set order that the rocks are in? Unlikely. Of course `MyBoxOfRocks` has to store the rocks somehow. Maybe it stores them in an array, and there is actually a set order of the rocks, even if it is artificially introduced for the sake of storage. On the other hand, maybe it doesn't. The point is once again, that's an implementation detail that you as the client of `MyBoxOfRocks` shouldn't have to worry about, and should *never* rely on. Presumably, `MyBoxOfRocks` wants to provide someway for clients to get at the rocks inside it. It could, once again, introduce an artificial order to the rocks; assign each rock an index and provide a method like `public Rock getRock(int aRockNumber)`. Or maybe it wants to put names on all the rocks and let you access it like `public Rock getRock(String aRockName)`. But maybe it's really just a box of rocks, and there's no names, no numbers, no way of identifying which rock you want, all you know is you want the rocks. Alright, let's try it with the visitor pattern. First out visitor interface (assume `Rock` is already defined somewhere, we don't care what it is or what it does):

```
public interface RockVisitor {
  public void visit(Rock aRock);
}
```

Easy. Now out `MyBoxOfRocks`

```
public class MyBoxOfRocks {

  private Rock[] fRocks;

  //... some kind of instantiation code

  public void foreach(RockVisitor aVisitor) {
    int length = fRocks.length;
    for (int i = 0; ilength; i++) {
      aVisitor.visit(fRocks[i]);
    }
  }

} // End class MyBoxOfRocks
```

Huh, what do you know, it does store them in an array. But what do we care now? We already wrote the visitor interface, and all we have to do now is implement it in some class which defines the actions to take for each rock, which we would have to do inside a `for` loop anyway. Besides, the array is private, our visitor doesn't have any access to that. And what if the implementor of `MyBoxOfRocks` did a little homework and found out that comparing a number to zero is infinitesimally faster than comparing it to a non zero. Infinitesimal, sure, but maybe when you're iterating over 10 million rocks, it makes a difference (**maybe**). So he decides to change the implementation:

```
public void foreach(RockVisitor aVisitor){
  int length = fRocks.length;
  for (int i=length-1; i=0; i--){
    aVisitor.visit(fRocks[i]);
  }
}
```

Now he's iterating backwards through the array and saving a (very) little time. He's changed the implementation because he found a better way. You didn't have to worry about finding the best way, and you didn't have to change your code because the implementation is encapsulated. And that's not the half of it. Maybe a new coder takes control of the project, and maybe this coder hates arrays and decides to totally change it:

```
public class MyBoxOfRocks {

  // This coder really likes Linked Lists
  private class RockNode {
    Rock iRock;
    RockNode iNext;

    RockNode(Rock aRock, RockNode aNext) {
       this.iRock = aRock;
       this.iNext = aNext;
    }
  } // end inner class RockNode

  private RockNode fFirstRock;

  // ... some instantiation code goes in here

  // Our new implementation
  public void foreach (RockVisitor aVisitor) {

    RockNode current = this.fFirstRock;
    // a null value indicates the list is ended
    while (current != null) {
      // have the visitor visit the current rock
      aVisitor.visit(current.iRock);
      // step to the next rock
      current = current.iNext;
    }
  }
}
```

Now maybe in this instance, linked lists were a poor idea, not as fast as a nice lean array and a `for` loop. On the other hand, you don't know what else this class is supposed to do. Maybe providing access to the rocks is only a small part of what it does, and linked lists fit in better with the rest of the requirements. In case I haven't said it enough yet, the point is that you as the client of `MyBoxOfRocks` don't have to worry about changes to the implementation, the visitor pattern protects you from it. I have one more trick up my sleeve. Maybe the implementor of `MyBoxOfRocks` notices that a lot of visitors are taking a really long time to visit each rock, and it's taking far too long for the `foreach` method to return because it has to wait for all visitors to finish. He decides it can't wait that long, and he also decides that some of these operations can probably be going on all at once. So he decides to do something about it, namely, this:

```
// Back to our backward-array model
public void foreach(RockVisitor aVisitor) {

  Thread t; // This should speed up the return

  int length = fRocks.length;
  for (int i = length-1; i = 0; i--) {
    final Rock curr = fRocks[i]
    t = new Thread() {
      public void run() {
        aVisitor.visit(curr);
      }
    }; // End anonymous Thread class
```

```
    t.start(); // Run the thread we just created.

    current = current.iNext;
  }
}
```

If you're familiar with threads, you'll understand what's going on here. If you're not, I'll quickly summarize: a Thread is basically something that can run "simultaneously" with other threads on the same machine. They don't *actually* run simultaneously of course, unless maybe you have a multi-processor machine, but as far as Java is concerned, they do. So for instance, when we created this new Thread called `t`, and defined what happens when the Thread is run (with the `run` method, naturally), we can then start the Thread a-running and it will start running, splitting cycles on the processor with other Threads, right away, it doesn't have to wait for the current method to return. Likewise, we can start it running and then continue on our own way immediately, without waiting for *it* to return. So with the above implementation, the only time we need to spend in this method is the time it takes to instantiate all the threads, *start* them running, and loop over the array; we don't have to wait for the visitor to actually visit each *Rock* before we can loop, we just loop right away, and the visitor does its thing on whatever CPU cycles it can swipe. The whole visiting process might take just as long (maybe even longer if it looses some cycles because of the multiple threads), but the thread from which `foreach` was invoked doesn't have to wait for it to finish, it can return from the method and be on it's way much faster. If you're confused about the use of the final Rock called `curr` in the above code, it's just a bit of a technicality for using anonymous classes: they can't access non final local variables. Even though `fRocks` doesn't fit into this category (it's not local, it's an instance variable), it `i` does. If you tried to remove this line and simply put `fRocks[i]` in the `run` method, it wouldn't compile. So what happens if you're the visitor, and you decide that you need to visit each rock one at a time? There's a number of reasons this might happen such as if your `visit` method changes your instance variables, or maybe it depends on the results of previous calls to `visit`. Well the implementation inside the `foreach` method is encapsulated from you, you don't know if it's using separate threads or not. Sure you could figure it out with some fancy debugging, or some clever printing to std out, but wouldn't it be nice if you didn't have to? And if you could be sure that if they change it in the next version, your code will still work properly? Well fortunately, Java provides the *synchronize* mechanism, which is basically an elaborate device for locking up blocks of code so that only one Thread can access them at a time. This won't conflict with the interests of the multi-threaded implementation, either, because the locked out thread still won't block the thread that created them, they'll just sit and wait patiently, only blocking code on itself. That's all well beyond the scope of this section, however, but be aware that it's available and probably worth looking into if you're going to be using synchronicity-sensitive visitors.

Implementation in Java

The following example is in the Java programming language[3]:

```
interface CarElementVisitor {
    void visit(Wheel wheel);
```

3 http://en.wikibooks.org/wiki/Java_Programming

```
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}


interface CarElement {
    void accept(CarElementVisitor visitor); // CarElements have to provide
 accept().
}


class Wheel implements CarElement {
    private String name;

    public Wheel(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void accept(CarElementVisitor visitor) {
        /*
         * accept(CarElementVisitor) in Wheel implements
         * accept(CarElementVisitor) in CarElement, so the call
         * to accept is bound at run time. This can be considered
         * the first dispatch. However, the decision to call
         * visit(Wheel) (as opposed to visit(Engine) etc.) can be
         * made during compile time since 'this' is known at compile
         * time to be a Wheel. Moreover, each implementation of
         * CarElementVisitor implements the visit(Wheel), which is
         * another decision that is made at run time. This can be
         * considered the second dispatch.
         */
        visitor.visit(this);
    }
}


class Engine implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}


class Body implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}


class Car implements CarElement {
    CarElement[] elements;

    public Car() {
        //create new Array of elements
        this.elements = new CarElement[] { new Wheel("front left"),
            new Wheel("front right"), new Wheel("back left") ,
            new Wheel("back right"), new Body(), new Engine() };
    }
```

```
    public void accept(CarElementVisitor visitor) {
        for(CarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}


class CarElementPrintVisitor implements CarElementVisitor {
    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 visit(Car car) {
        System.out.println("Visiting car");
    }
}


class CarElementDoVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }

    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}


public class VisitorDemo {
    public static void main(String[] args) {
        CarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

Implementation in D

The following example is in the D programming language[4]:

```
import std.stdio;
```

4 http://en.wikibooks.org/wiki/D_(The_Programming_Language)

```
import std.string;

interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body bod);
    void visitCar(Car car);
}

interface CarElement{
    void accept(CarElementVisitor visitor);
}

class Wheel : CarElement {
    private string name;
    this(string name) {
        this.name = name;
    }
    string getName() {
        return name;
    }
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine : CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body : CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car {
    CarElement[] elements;
    public CarElement[] getElements(){
        return elements;
    }
    public this() {
        elements =
        [
            cast(CarElement) new Wheel("front left"),
            cast(CarElement) new Wheel("front right"),
            cast(CarElement) new Wheel("back left"),
            cast(CarElement) new Wheel("back right"),
            cast(CarElement) new Body(),
            cast(CarElement) new Engine()
        ];
    }
}

class CarElementPrintVisitor : CarElementVisitor {
    public void visit(Wheel wheel) {
        writefln("Visiting "~ wheel.getName() ~ " wheel");
    }
    public void visit(Engine engine) {
        writefln("Visiting engine");
    }
    public void visit(Body bod) {
        writefln("Visiting body");
    }
    public void visitCar(Car car) {
        writefln("\nVisiting car");
```

```
        foreach(CarElement element ; car.elements) {
            element.accept(this);
        }
        writefln("Visited car");
    }
}

class CarElementDoVisitor : CarElementVisitor {
    public void visit(Wheel wheel) {
        writefln("Kicking my "~ wheel.name);
    }
    public void visit(Engine engine) {
        writefln("Starting my engine");
    }
    public void visit(Body bod) {
        writefln("Moving my body");
    }
    public void visitCar(Car car) {
        writefln("\nStarting my car");
        foreach(CarElement carElement ; car.getElements()) {
            carElement.accept(this);
        }
        writefln("Started car");
    }
}

void main(){
    Car car = new Car;
    CarElementVisitor printVisitor = new CarElementPrintVisitor;
    CarElementVisitor doVisitor = new CarElementDoVisitor;
    printVisitor.visitCar(car);
    doVisitor.visitCar(car);
}
```

Implementation in C#

The following example is an example in the C# programming language[5]:

```
using System;

namespace VisitorPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            var car = new Car();
            CarElementVisitor printVisitor = new CarElementPrintVisitor();
            CarElementVisitor doVisitor = new CarElementDoVisitor();
            printVisitor.visitCar(car);
            doVisitor.visitCar(car);
        }
    }

    public interface CarElementVisitor
    {
        void visit(Wheel wheel);
        void visit(Engine engine);
        void visit(Body body);
        void visitCar(Car car);
    }
    public interface CarElement
    {
```

5 http://en.wikibooks.org/wiki/C_Sharp_Programming

```
        void accept(CarElementVisitor visitor); // CarElements have to provide
 accept().
    }
    public class Wheel : CarElement
    {

        public String name { get; set; }

        public void accept(CarElementVisitor visitor)
        {
            visitor.visit(this);
        }
    }

    public class Engine : CarElement
    {
        public void accept(CarElementVisitor visitor)
        {
            visitor.visit(this);
        }
    }

    public class Body : CarElement
    {
        public void accept(CarElementVisitor visitor)
        {
            visitor.visit(this);
        }
    }

    public class Car
    {
        public CarElement[] elements { get; private set; }

        public Car()
        {
            elements = new CarElement[]
          { new Wheel{name = "front left"}, new Wheel{name = "front right"},
            new Wheel{name = "back left"} , new Wheel{name="back right"},
            new Body(), new Engine() };
        }
    }

    public class CarElementPrintVisitor : CarElementVisitor
    {
        public void visit(Wheel wheel)
        {
            Console.WriteLine("Visiting " + wheel.name + " wheel");
        }
        public void visit(Engine engine)
        {
            Console.WriteLine("Visiting engine");
        }
        public void visit(Body body)
        {
            Console.WriteLine("Visiting body");
        }

        public void visitCar(Car car)
        {
            Console.WriteLine("\nVisiting car");
            foreach (var element in car.elements)
            {
                element.accept(this);
            }
            Console.WriteLine("Visited car");
        }
    }
```

```
    public class CarElementDoVisitor : CarElementVisitor
    {
        public void visit(Wheel wheel)
        {
            Console.WriteLine("Kicking my " + wheel.name);
        }
        public void visit(Engine engine)
        {
            Console.WriteLine("Starting my engine");
        }
        public void visit(Body body)
        {
            Console.WriteLine("Moving my body");
        }
        public void visitCar(Car car)
        {
            Console.WriteLine("\nStarting my car");
            foreach (var element in car.elements)
            {
                element.accept(this);
            }
        }

    }
}
```

Implementation in Lisp

```
(defclass auto ()
  ((elements :initarg :elements)))

(defclass auto-part ()
  ((name :initarg :name :initform "unnamed-car-part")))

(defmethod print-object ((p auto-part) stream)
  (print-object (slot-value p 'name) stream))

(defclass wheel (auto-part) ())

(defclass body (auto-part) ())

(defclass engine (auto-part) ())

(defgeneric traverse (function object other-object))

(defmethod traverse (function (a auto) other-object)
  (with-slots (elements) a
    (dolist (e elements)
      (funcall function e other-object))))

;; do-something visitations

;; catch all
(defmethod do-something (object other-object)
  (format t "don't know how ~s and ~s should interact~%" object other-object))

;; visitation involving wheel and integer
(defmethod do-something ((object wheel) (other-object integer))
  (format t "kicking wheel ~s ~s times~%" object other-object))

;; visitation involving wheel and symbol
(defmethod do-something ((object wheel) (other-object symbol))
  (format t "kicking wheel ~s symbolically using symbol ~s~%" object
 other-object))
```

```
(defmethod do-something ((object engine) (other-object integer))
  (format t "starting engine ~s ~s times~%" object other-object))

(defmethod do-something ((object engine) (other-object symbol))
  (format t "starting engine ~s symbolically using symbol ~s~%" object
 other-object))

(let ((a (make-instance 'auto
                        :elements `(,(make-instance 'wheel :name
 "front-left-wheel")
                                    ,(make-instance 'wheel :name
 "front-right-wheel")
                                    ,(make-instance 'wheel :name
 "rear-right-wheel")
                                    ,(make-instance 'wheel :name
 "rear-right-wheel")
                                    ,(make-instance 'body :name "body")
                                    ,(make-instance 'engine :name "engine")))))
  ;; traverse to print elements
  ;; stream *standard-output* plays the role of other-object here
  (traverse #'print a *standard-output*)

  (terpri) ;; print newline

  ;; traverse with arbitrary context from other object
  (traverse #'do-something a 42)

  ;; traverse with arbitrary context from other object
  (traverse #'do-something a 'abc))
```

25 Print version

The model-view-controller (MVC) pattern is an architectural pattern used primarily in creating Graphic User Interfaces[1] (GUIs). The major premise of the pattern is based on modularity and it is to separate three different aspects of the GUI: the data (model), the visual representation of the data (view), and the interface between the view and the model (controller). The primary idea behind keeping these three components separate is so that each one is as independent of the others as possible, and changes made to one will not affect changes made to the others. In this way, for instance, the GUI can be updated with a new look or visual style without having to change the data model or the controller. Newcomers will probably see this MVC pattern as wasteful, mainly because you are working with many extra objects at runtime, when it seems like one giant object will do. But the secret to the MVC pattern is not writing the code, but in maintaining it, and allowing people to modify the code without changing much else. Also, keep in mind, that different developers have different strengths and weaknesses, so team building around MVC is easier. Imagine a View Team that is responsible for great views, a Model Team that knows a lot about data, and a Controller Team that see the big picture of application flow, handing requests, working with the model, and selecting the most appropriate next view for that client. One of the great advantages of the Model-View-Controller Pattern is the ability to reuse the application's logic (which is implemented in the model) when implementing a different view. A good example is found in web development, where a common task is to implement an external API inside of an existing piece of software. If the MVC pattern has cleanly been followed, this only requires modification to the controller, which can have the ability to render different types of views dependent on the content type requested by the user agent.

25.1 Model

Generally, the model is constructed first. This doesn't necessarily need to be anything special, it is just an object the way you would normally make an object, with properties that can be configured and queried using getters and setters. The important thing to remember about the model is that every property that is going to show up in the view needs to support property listeners. This will allow the controller to update the view when the model changes.

1 http://en.wikibooks.org//en.wikipedia.org/wiki/Graphical_user_interface

25.2 View

This can often be constructed after the model. Frequently, each property in the model will have its own component in the GUI, called an *editor*. Certain property types often have standard associated editor types. For instance, a property whose value in an unconstrained text string might be a text field, or a text box. Numeric properties will often use a text field as well, but in this case, the controller will need to do some checking on the entered value to make sure it is really a number. Properties with a small predefined set of allowed values will often use a combo box[2] to allow the users to select one of the predefined values. More complex properties will likely require more complex editors. Like the model, the view will generally need to support listeners so that the controller can update the model based on the user input. Different languages handle this in different ways. Some languages allow callback functions to be associated with editors, which get called when the value is changed. Other languages allow you to add listeners directly to the component, which can be notified when the value in the editor changes, when it is clicked on, or when it looses focus, etc.

25.3 Controller

The controller frequently isn't an actual object, but a collection of methods and listeners, often built in to both the model and the view. The general pattern is that both the model and the view have a certain interface which provide accessibility to their data values or editor respectively. For instance, for a property called `Title`, the model may have methods `getTitle` and `setTitle`, while the view might have `setTitleFieldText` and `getTitleFieldText`. Alternatively, the view may simply provide accessor methods to its editor components directly, so you might instead have `getTitleField().setText()` and `getTitleField().getText()`. The controller is then designed to "plug into" each of these interfaces, and pass data between them. When the controller is notified through one of its listeners that either the model or the view has changed, it may validate the new value (particularly, if the value is coming from the view) and then pass it along to the other component.

25.3.1 Validation

When possible, it is usually best to allow the model to do all the necessary validation of values, so that any changes to the allowed values, or changes simply to the validation process, only need to be made in one place. However, in some languages under certain circumstances, this may not be possible. For instance, if a numeric property is being edited with a text field, then the value of the property passed to the controller by the view will be text, not a number. In this case, the model could be made to have an additional method that takes text as the input value for this property, but more likely, the controller will do the initial parsing of the text to get the numeric value, and then pass it on to the model to do further validation (for instance, bounds checking). When either the controller or the model determines that a passed in value is invalid, the controller will need to tell the view

2 `http://en.wikibooks.org//en.wikipedia.org/wiki/Combo_box`

that the value is invalid. In some cases, the view may then issue an error dialog or other notification, or it may simply revert the value in its editor to the older valid value.

Implementation in Java

In Java, we can implement a fat client GUI. In this case, we can use:

- JavaBeans[3] to implement the model and,
- SWT or Java Swings[4] to implement the view and the controller.

However, in Java EE, we can also implement a web application. In this case, we can use:

- EJB entity[5] to implement the model,
- JSP[6] to implement the view and,
- EJB session[7] to implement the controller.

Hidden category:

- No references for citations[8]

3 http://en.wikibooks.org/wiki/Java_Programming/JavaBeans
4 http://en.wikibooks.org/wiki/Java_Swings
5 http://en.wikibooks.org/wiki/J2EE_Programming/Enterprise_JavaBeans
6 http://en.wikibooks.org/wiki/J2EE_Programming/JavaServer_Pages
7 http://en.wikibooks.org/wiki/J2EE_Programming/Enterprise_JavaBeans
8 http://en.wikibooks.org/wiki/Category:No_references_for_citations

26 Contributors

Edits	User
1	Ftiercel[1]

1 http://en.wikibooks.org/wiki/User:Ftiercel

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[2]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

2 Chapter 27 on page 271

1	**Doktor**[3] ***Mandrake***[4], **Doktor**[5] ***Mandrake***[6]	GFDL
2	Amnon Eden[7]	
3	Giacomo Ritucci[8], Giacomo Ritucci[9]	GFDL
4	Trashtoy[10], Trashtoy[11]	
5	User:Giacomo Ritucci[12], User:Giacomo Ritucci[13]	GFDL
6	Spischot[14], Spischot[15]	PD
7	Econt[16], Econt[17]	PD
8	Econt[18], Econt[19]	PD
9	Spischot[20], Spischot[21]	PD
10	Spischot[22], Spischot[23]	PD
11	Spischot[24], Spischot[25]	PD
12	Spischot[26], Spischot[27]	PD
13	Spischot[28], Spischot[29]	PD
14	Spischot[30], Spischot[31]	PD

3 http:////commons.wikimedia.org/wiki/User:DoktorMandrake
4 http:////commons.wikimedia.org/wiki/User_talk:DoktorMandrake
5 http:///wiki/User:DoktorMandrake
6 http:///wiki/User_talk:DoktorMandrake
7 http:////en.wikipedia.org/wiki/User:Edenphd
8 http:////commons.wikimedia.org/wiki/User:Giacomo_Ritucci
9 http:///wiki/User:Giacomo_Ritucci
10 http:////commons.wikimedia.org/wiki/User:Trashtoy
11 http:///wiki/User:Trashtoy
12 http:////commons.wikimedia.org/wiki/User:Giacomo_Ritucci
13 http:///wiki/User:Giacomo_Ritucci
14 http:////commons.wikimedia.org/wiki/User:Spischot
15 http:///wiki/User:Spischot
16 http:////commons.wikimedia.org/wiki/User:Econt
17 http:///wiki/User:Econt
18 http:////commons.wikimedia.org/wiki/User:Econt
19 http:///wiki/User:Econt
20 http:////commons.wikimedia.org/wiki/User:Spischot
21 http:///wiki/User:Spischot
22 http:////commons.wikimedia.org/wiki/User:Spischot
23 http:///wiki/User:Spischot
24 http:////commons.wikimedia.org/wiki/User:Spischot
25 http:///wiki/User:Spischot
26 http:////commons.wikimedia.org/wiki/User:Spischot
27 http:///wiki/User:Spischot
28 http:////commons.wikimedia.org/wiki/User:Spischot
29 http:///wiki/User:Spischot
30 http:////commons.wikimedia.org/wiki/User:Spischot
31 http:///wiki/User:Spischot

15	• Composite_UML_class_diagram.svg[32]: Trashtoy[33] • derivative work: « Aaron Rotenberg[34] « Talk[35] « , • Composite_UML_class_diagram.svg[36]: Trashtoy[37] • derivative work: « Aaron Rotenberg[38] « Talk[39] «	

32 http:////commons.wikimedia.org/wiki/File:Composite_UML_class_diagram.svg
33 http:////commons.wikimedia.org/wiki/User:Trashtoy
34 http:////commons.wikimedia.org/wiki/User:Aaron_Rotenberg
35 http:////commons.wikimedia.org/wiki/User_talk:Aaron_Rotenberg
36 http:///wiki/File:Composite_UML_class_diagram.svg
37 http:///wiki/User:Trashtoy
38 http:///wiki/User:Aaron_Rotenberg
39 http:///wiki/User_talk:Aaron_Rotenberg

27 Licenses

27.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”. * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's “contributor version”.

A contributor's “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

27.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

27.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.