Computer Science Design Patterns

en.wikibooks.org
On the 28th of April 2012 the contents of the English as well as German Wikibooks and Wikipedia projects were licensed under Creative Commons Attribution-ShareAlike 3.0 Unported license. A URI to this license is given in the list of figures on page 349. If this document is a derived work from the contents of one of these projects and the content was still licensed by the project under this license at the time of derivation this document has to be licensed under the same, a similar or a compatible license, as stated in section 4b of the license. The list of contributors is included in chapter Contributors on page 325. The licenses GPL, LGPL and GFDL are included in chapter Licenses on page 355, since this book and/or parts of it may or may not be licensed under one or more of these licenses, and thus require inclusion of these licenses. The licenses of the figures are given in the list of figures on page 349. This PDF was generated by the \LaTeX typesetting software. The \LaTeX source code is included as an attachment (source.7z.txt) in this PDF file. To extract the source from the PDF file, you can use the pdfdetach tool including in the poppler suite, or the http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/ utility. Some PDF viewers may also let you save the attachment to a file. After extracting it from the PDF file you have to rename it to source.7z. To uncompressed the resulting archive we recommend the use of http://www.7-zip.org/. The \LaTeX source itself was generated by a program written by Dirk Hünniger, which is freely available under an open source license from http://de.wikibooks.org/wiki/Benutzer:Dirk_Huenniger/wb2pdf.
Contents

1 Abstract Factory 3
  1.1 Definition ............................................. 3
  1.2 Usage ................................................... 4
  1.3 Structure ................................................. 5
  1.4 Implementations ......................................... 7

2 Adapter 21
  2.1 Examples ................................................. 22
  2.2 Cost ..................................................... 22
  2.3 Advices .................................................. 22
  2.4 Implementation .......................................... 22

3 Bridge 29
  3.1 Examples ................................................. 30
  3.2 Cost ..................................................... 30
  3.3 Advices .................................................. 31
  3.4 Implementation .......................................... 31

4 Builder 45
  4.1 Examples ................................................. 47
  4.2 Cost ..................................................... 47
  4.3 Advices .................................................. 47
  4.4 Implementations .......................................... 47

5 Chain of responsibility 67

6 Command 87
  6.1 Scope .................................................... 87
  6.2 Purpose .................................................. 87
  6.3 Intent .................................................... 87
  6.4 Applicability ............................................ 87
  6.5 Structure ................................................ 87
  6.6 Consequences ............................................ 88
  6.7 Examples ................................................ 88
  6.8 Cost ...................................................... 88
  6.9 Advices .................................................. 89
  6.10 Implementations ......................................... 89
  6.11 Operations .............................................. 90

7 Composite 101
  7.1 Examples ............................................... 103
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Cost</td>
<td>103</td>
</tr>
<tr>
<td>7.3</td>
<td>Advices</td>
<td>104</td>
</tr>
<tr>
<td>7.4</td>
<td>Implementations</td>
<td>104</td>
</tr>
<tr>
<td>8</td>
<td>Decorator</td>
<td>111</td>
</tr>
<tr>
<td>8.1</td>
<td>Examples</td>
<td>111</td>
</tr>
<tr>
<td>8.2</td>
<td>Cost</td>
<td>111</td>
</tr>
<tr>
<td>8.3</td>
<td>Advises</td>
<td>111</td>
</tr>
<tr>
<td>8.4</td>
<td>Implementations</td>
<td>111</td>
</tr>
<tr>
<td>8.5</td>
<td>Coffee making scenario</td>
<td>112</td>
</tr>
<tr>
<td>8.6</td>
<td>First Example (window/scrolling scenario)</td>
<td>115</td>
</tr>
<tr>
<td>8.7</td>
<td>Second Example (coffee making scenario)</td>
<td>117</td>
</tr>
<tr>
<td>8.8</td>
<td>Window System</td>
<td>119</td>
</tr>
<tr>
<td>8.9</td>
<td>Coffee example</td>
<td>120</td>
</tr>
<tr>
<td>8.10</td>
<td>External links</td>
<td>120</td>
</tr>
<tr>
<td>9</td>
<td>Facade</td>
<td>121</td>
</tr>
<tr>
<td>9.1</td>
<td>Examples</td>
<td>121</td>
</tr>
<tr>
<td>9.2</td>
<td>Cost</td>
<td>121</td>
</tr>
<tr>
<td>9.3</td>
<td>Advises</td>
<td>121</td>
</tr>
<tr>
<td>9.4</td>
<td>Implementations</td>
<td>122</td>
</tr>
<tr>
<td>10</td>
<td>Factory method</td>
<td>131</td>
</tr>
<tr>
<td>10.1</td>
<td>Basic Implementation of the Factory Pattern</td>
<td>131</td>
</tr>
<tr>
<td>10.2</td>
<td>Factory Pattern Implementation of the Alphabet</td>
<td>132</td>
</tr>
<tr>
<td>10.3</td>
<td>The Factory Pattern and Parametric Polymorphism</td>
<td>138</td>
</tr>
<tr>
<td>10.4</td>
<td>Examples</td>
<td>139</td>
</tr>
<tr>
<td>10.5</td>
<td>Cost</td>
<td>139</td>
</tr>
<tr>
<td>10.6</td>
<td>Good practices</td>
<td>140</td>
</tr>
<tr>
<td>10.7</td>
<td>Implementation</td>
<td>140</td>
</tr>
<tr>
<td>10.8</td>
<td>Example with pizza</td>
<td>146</td>
</tr>
<tr>
<td>10.9</td>
<td>Another example with image</td>
<td>147</td>
</tr>
<tr>
<td>11</td>
<td>Flyweight</td>
<td>155</td>
</tr>
<tr>
<td>11.1</td>
<td>Examples</td>
<td>155</td>
</tr>
<tr>
<td>11.2</td>
<td>Cost</td>
<td>156</td>
</tr>
<tr>
<td>11.3</td>
<td>Advises</td>
<td>157</td>
</tr>
<tr>
<td>11.4</td>
<td>Implementations</td>
<td>157</td>
</tr>
<tr>
<td>12</td>
<td>Interpreter</td>
<td>159</td>
</tr>
<tr>
<td>12.1</td>
<td>Examples</td>
<td>159</td>
</tr>
<tr>
<td>12.2</td>
<td>Cost</td>
<td>159</td>
</tr>
<tr>
<td>12.3</td>
<td>Advises</td>
<td>160</td>
</tr>
<tr>
<td>12.4</td>
<td>Implementations</td>
<td>160</td>
</tr>
<tr>
<td>13</td>
<td>Iterator</td>
<td>175</td>
</tr>
<tr>
<td>13.1</td>
<td>Examples</td>
<td>175</td>
</tr>
<tr>
<td>13.2</td>
<td>Cost</td>
<td>175</td>
</tr>
<tr>
<td>13.3</td>
<td>Advises</td>
<td>175</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>13.4</td>
<td>Implementations</td>
<td>175</td>
</tr>
<tr>
<td>13.5</td>
<td>References</td>
<td>188</td>
</tr>
<tr>
<td>14</td>
<td>Mediator</td>
<td>189</td>
</tr>
<tr>
<td>14.1</td>
<td>Participants</td>
<td>189</td>
</tr>
<tr>
<td>14.2</td>
<td>Examples</td>
<td>189</td>
</tr>
<tr>
<td>14.3</td>
<td>Cost</td>
<td>189</td>
</tr>
<tr>
<td>14.4</td>
<td>Advises</td>
<td>190</td>
</tr>
<tr>
<td>14.5</td>
<td>Implementations</td>
<td>190</td>
</tr>
<tr>
<td>15</td>
<td>Memento</td>
<td>195</td>
</tr>
<tr>
<td>15.1</td>
<td>C# Example</td>
<td>196</td>
</tr>
<tr>
<td>15.2</td>
<td>Another way to implement memento in C#</td>
<td>197</td>
</tr>
<tr>
<td>16</td>
<td>Model–view–controller</td>
<td>205</td>
</tr>
<tr>
<td>16.1</td>
<td>Model</td>
<td>205</td>
</tr>
<tr>
<td>16.2</td>
<td>View</td>
<td>206</td>
</tr>
<tr>
<td>16.3</td>
<td>Controller</td>
<td>206</td>
</tr>
<tr>
<td>17</td>
<td>Observer</td>
<td>209</td>
</tr>
<tr>
<td>17.1</td>
<td>Scope</td>
<td>209</td>
</tr>
<tr>
<td>17.2</td>
<td>Purpose</td>
<td>209</td>
</tr>
<tr>
<td>17.3</td>
<td>Intent</td>
<td>209</td>
</tr>
<tr>
<td>17.4</td>
<td>Applicability</td>
<td>209</td>
</tr>
<tr>
<td>17.5</td>
<td>Structure</td>
<td>209</td>
</tr>
<tr>
<td>17.6</td>
<td>Consequences</td>
<td>210</td>
</tr>
<tr>
<td>17.7</td>
<td>Implementation</td>
<td>210</td>
</tr>
<tr>
<td>17.8</td>
<td>Related pattern</td>
<td>210</td>
</tr>
<tr>
<td>17.9</td>
<td>Description</td>
<td>210</td>
</tr>
<tr>
<td>17.10</td>
<td>Examples</td>
<td>211</td>
</tr>
<tr>
<td>17.11</td>
<td>Cost</td>
<td>211</td>
</tr>
<tr>
<td>17.12</td>
<td>Advises</td>
<td>212</td>
</tr>
<tr>
<td>17.13</td>
<td>Implementations</td>
<td>212</td>
</tr>
<tr>
<td>17.14</td>
<td>Traditional Method</td>
<td>214</td>
</tr>
<tr>
<td>17.15</td>
<td>Using Events</td>
<td>215</td>
</tr>
<tr>
<td>17.16</td>
<td>Handy implementation</td>
<td>216</td>
</tr>
<tr>
<td>17.17</td>
<td>Built-in support</td>
<td>218</td>
</tr>
<tr>
<td>18</td>
<td>Prototype</td>
<td>227</td>
</tr>
<tr>
<td>18.1</td>
<td>Structure</td>
<td>227</td>
</tr>
<tr>
<td>18.2</td>
<td>Rules of thumb</td>
<td>227</td>
</tr>
<tr>
<td>18.3</td>
<td>Advices</td>
<td>228</td>
</tr>
<tr>
<td>18.4</td>
<td>Implementations</td>
<td>228</td>
</tr>
<tr>
<td>19</td>
<td>Proxy</td>
<td>239</td>
</tr>
<tr>
<td>20</td>
<td>Singleton</td>
<td>251</td>
</tr>
<tr>
<td>20.1</td>
<td>Refresh Java object creation</td>
<td>251</td>
</tr>
<tr>
<td>20.2</td>
<td>Singleton &amp; Multi Threading</td>
<td>253</td>
</tr>
</tbody>
</table>
## Contents

20.3 Examples .................................................. 254
20.4 Cost ........................................................... 254
20.5 Advises ....................................................... 255
20.6 Implementation ............................................. 255
20.7 Traditional simple way using synchronization .......... 255
20.8 Initialization on Demand Holder Idiom .................. 256
20.9 The Enum-way .............................................. 256

21 State ............................................................ 265
  21.1 References .................................................. 274

22 Strategy ......................................................... 275
  22.1 Scope ......................................................... 275
  22.2 Purpose ...................................................... 275
  22.3 Intent ........................................................ 275
  22.4 Applicability ................................................ 275
  22.5 Structure ..................................................... 275
  22.6 Consequences .............................................. 276
  22.7 Implementation .......................................... 276
  22.8 Related patterns .......................................... 276
  22.9 Description ............................................... 276
  22.10 Examples .................................................... 280
  22.11 Cost .......................................................... 280
  22.12 Advises ...................................................... 281
  22.13 Implementations ......................................... 281

23 Template method ............................................... 299

24 Visitor ........................................................ 307
  24.1 An example implementation of Visitor Pattern: String .. 308
  24.2 Another example implementation of Visitor Pattern: Rock .. 309
  24.3 Examples ..................................................... 312
  24.4 Cost ........................................................... 312
  24.5 Advises ....................................................... 313
  24.6 Implementations ........................................... 313

25 Development stages ........................................... 323

26 Contributors .................................................. 325

List of Figures .................................................. 349

27 Licenses ......................................................... 355
  27.1 GNU GENERAL PUBLIC LICENSE ............................ 355
  27.2 GNU Free Documentation License ........................ 356
  27.3 GNU Lesser General Public License ....................... 357
1 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.

1.1 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".
1.2 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.
1.3 Structure

1.3.1 Class diagram

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.
1.3.2 Lepus3 chart (legend)

Figure 2

1.3.3 UML diagram

Figure 3
1.4 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.

```csharp
using System;
using System.Configuration;

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

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

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

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

    public class ApplicationRunner {
        static IGUlnFactory CreateOsSpecificFactory() { // Executes second
            // Contents of App.Config associated with this C# project
            // <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();
            }
        }
    }
```
Abstract Factory

```cpp
/* 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();
    }
};

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

```cpp
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;
}
```

/* GUIFactory example */

```java
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();
    }
}

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();
}

/* GUIFactory example -- */

#import <Foundation/Foundation.h>

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

@interface WinButton : NSObject <Button>
@end

@interface OSXButton : NSObject <Button>
@end

@protocol GUIFactory
- (id<Button>)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
- (id<Button>)createButton {
    return [[[WinButton alloc] init] autorelease];
}
@end

@implementation OSXFactory
- (id<Button>)createButton {
    return [[[OSXButton alloc] init] autorelease];
}
@end

@implementation Application
- (id)initWithGUIFactory:(id<GUIFactory>)factory {
    if (self = [super init]) {
        id button = [factory createButton];
        [button paint];
    }
    return self;
}

+ (id<GUIFactory>)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;
}

--[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()

Abstract Factory

This abstract factory creates books.

```lua
print("I'm a fancy Windows button!")
-- Create the OSXGuiFactory Class
function OSXGuiFactory:new()
  local inst = {}
  setmetatable(inst, mt)
  return inst
end
function OSXGuiFactory:CreateButton()
  return OSXButton:new()
end
-- Create the WinGuiFactory Class
function WinGuiFactory:new()
  local inst = {}
  setmetatable(inst, mt)
  return inst
end
function WinGuiFactory:CreateButton()
  return WinButton:new()
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()
```

```java
/*
 * 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;
    }
}

abstract class AbstractBook {
    abstract function getAuthor();
    abstract function getTitle();
}

abstract class AbstractMySQLBook extends AbstractBook {
    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 extends AbstractBook {
    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 ($rand_num == 1) {
            $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('');
writeln('END TESTING ABSTRACT FACTORY PATTERN');
writeln('');
Implementations

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/>";
}

This abstract factory creates musicians.
// concrete implementation
class DrummerFactory extends MusicianAbstractFactory {
def create(): Musician = new Drummer()
}
class GuitarPlayerFactory extends MusicianAbstractFactory {
def create(): Musician = new GuitarPlayer()
}
class GuitarPlayer extends Musician {
def play(song: String) {
println(s"I'm guitar player and I play $song")
}
}
class Drummer extends Musician {
def play(song: String) {
println(s"I'm drummer and I play '$song'")
}
}
object FactoryCreator {
def getFactory(config: Boolean): MusicianAbstractFactory =
if (config) {
new DrummerFactory
} else {
new GuitarPlayerFactory
}
}

// interfaces to import
trait Musician {
def play(song: String)
}
trait MusicianAbstractFactory {
def create(): Musician
}

import implementation.FactoryCreator
object AbstractFactoryTest {
def readFromConfig(): Boolean = true

15


def main(args: Array[String]) {
    val factory = FactoryCreator.getFactory(readFromConfig())
    val musician = factory.create()
    musician.play("Highway to hell")
} 

from abc import ABC, abstractmethod
from sys import platform

class Button(ABC):
    @abstractmethod
    def paint(self):
        pass

class LinuxButton(Button):
    def paint(self):
        return "Render a button in a Linux style"

class WindowsButton(Button):
    def paint(self):
        return "Render a button in a Windows style"

class MacOSButton(Button):
    def paint(self):
        return "Render a button in a MacOS style"

class GUIFactory(ABC):
    @abstractmethod
    def create_button(self):
        pass

class LinuxFactory(GUIFactory):
    def create_button(self):
        return LinuxButton()

class WindowsFactory(GUIFactory):
    def create_button(self):
        return WindowsButton()

class MacOSFactory(GUIFactory):
    def create_button(self):
        return MacOSButton()

if platform == "linux":
    factory = LinuxFactory()
elif platform == "darwin":
    factory = MacOSFactory()
elif platform == "win32":
    factory = WindowsFactory()
else:
    raise NotImplementedError("Not implemented for your platform: {platform}")

button = factory.create_button()
result = button.paint()
print(result)

Alternative implementation using the classes themselves as factories:

```python
from abc import ABC, abstractmethod
from sys import platform

class Button(ABC):
    @abstractmethod
    def paint(self):
        pass

class LinuxButton(Button):
    def paint(self):
        return "Render a button in a Linux style"

class WindowsButton(Button):
    def paint(self):
        return "Render a button in a Windows style"

class MacOSButton(Button):
    def paint(self):
        return "Render a button in a MacOS style"

if platform == "linux":
    factory = LinuxButton
elif platform == "darwin":
    factory = MacOSButton
elif platform == "win32":
    factory = WindowsButton
else:
    raise NotImplementedError(f"Not implemented for your platform: {platform}\n")

button = factory()
result = button.paint()
print(result)
```

Another example:

```python
import platform

class GuiFactory():
    """Abstract Factory""

    @classmethod
    def getFactory(Class):
        if platform.system() == "Windows":
            return WinFactory()
        elif platform.system() == "OSX":
            return OSXFactory()
        elif platform.system() == "Linux":
            return LinuxFactory()
        else:
            raise NotImplementedError(f"Not implemented for your platform: {platform}\n")

class Button:
    """Abstract Product""
    def paint(self):
        raise NotImplementedError
```
Abstract Factory

```python
@classmethod
def getButton(Class):
    return Class.Button()

class WinFactory(GuiFactory):
    """Concrete Factory""

class Button(GuiFactory.Button):
    """Concrete Product"
    def paint(self):
        print "I am a Windows button"

class LinuxFactory(GuiFactory):
    """Concrete Factory""

class Button(GuiFactory.Button):
    """Concrete Product"
    def paint(self):
        print "I am a Linux button"

class OSXFactory(GuiFactory):
    """Concrete Factory""

class Button(GuiFactory.Button):
    """Concrete Product"
    def paint(self):
        print "I am a OSX button"

def main():
    """Application"
    factory = GuiFactory.getFactory()
    factory.getButton().paint()

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

```delphi
program AbstractFactory;

{$APPTYPE CONSOLE}
{$R *.res}

uses
    System.SysUtils;

type
    (* abstract factory *)
    TAbstractFactory = class abstract
        procedure Paint(); virtual; abstract;
    end;

    (* abstract product *)
    TGUConnectionFactory = class abstract
        function CreateButton(): TAbstractFactory; virtual; abstract;
    end;

    (* concrete product *)
    TOSXButton = class(TAbstractFactory)
        public
            procedure Paint(); override;
        end;

    (* concrete product *)
    TWinButton = class(TAbstractFactory)
        public
```
procedure Paint(); override;
end;

(* concrete factory *)
TOSXFactory = class(TGUIFactory)
public
  function CreateButton(): TAbstractFactory; override;
end;

(* concrete factory *)
TWinFactory = class(TGUIFactory)
public
  function CreateButton(): TAbstractFactory; override;
end;

(* client *)
TApplication = class
public
  constructor Create(factory: TGUIFactory);
end;

(* Just for OOP style use class. This function is to return button factory only. *)
TApplicationRunner = class
public
  class function CreateOsSpecificFactory: TGUIFactory;
end;

{ TOSXButton }
procedure TOSXButton.Paint;
begin
  WriteLn('I`m an OSXButton');
end;

{ TWinButton }
procedure TWinButton.Paint;
begin
  WriteLn('I`m a WinButton');
end;

{ TOSXFactory }
function TOSXFactory.CreateButton: TAbstractFactory;
begin
  Result := TOSXButton.Create;
end;

{ TWinFactory }
function TWinFactory.CreateButton: TAbstractFactory;
begin
  Result := TWinButton.Create;
end;

{ TApplication }
constructor TApplication.Create(factory: TGUIFactory);
var
  button: TAbstractFactory;
begin
  button := factory.CreateButton;
  button.Paint;
end;

{ TApplicationRunner }
class function TApplicationRunner.CreateOsSpecificFactory: TGUIFactory;
var
  sysType: string;
begin
WriteLn('Enter OS type (Win: Windows, OSX: MacOS X)');
ReadLn(sysType);
if (sysType = 'Win') then
  Result := TWinFactory.Create
else
  Result := TOSXFactory.Create
end;

var
  App: TApplication;
begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
    App := TApplication.Create(TApplicationRunner.CreateOsSpecificFactory);
    WriteLn('#13#10 + 'Press any key to continue...');
    ReadLn;
    App.Free;
  except
    on E: Exception do
      WriteLn(E.ClassName, ' : ', E.Message);
  end;
end.
2 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)

EmployeeAdapter

MailingClient
+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.
2.1 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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 Removal

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

2.3 Advices

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

2.4 Implementation

2.4.1 Object Adapter

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

\(^1\) [https://github.com/corbanbrook/webgl-2d](https://github.com/corbanbrook/webgl-2d)
/**
 * 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 CompanyB Employees {
    /**
     * Retrieve the employee information from the LDAP.
     * @param distinguishedName The LDAP distinguishedName.
     * @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 department The employee department.
     * @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, CompanyAAdapter:

```java
public class CompanyAAdapter implements EmployeeBrowser {
    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, CompanyBAdapter:

```java
public class CompanyBAdapter implements EmployeeBrowser {
    public Employee getEmployee(String direction, String division, String department, String service, String firstName, String lastName) {
        String distinguishedName = "ov1 = " + direction + ", ov2 = " + division + " AND employee.company= 'COMPANY B' AND employee.department = " + department + " AND employee.service = " + service + " AND employee.firstName = " + firstName + " AND employee.lastName = " + lastName;
        CompanyBEmployees companyBEmployees = new CompanyBEmployees();
        return companyBEmployees.getEmployee(distinguishedName);
    }
}
```
2.4.2 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
```

```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()
```

```scala
trait Socket220V {
  def plug220()
}
trait Socket19V {
  def plug19()
}

class Laptop extends Socket19V {
  def plug19() {
    println("Charging....")
  }
}

class LaptopAdapter(laptop: Laptop) extends Socket220V {
  def plug220() {
    println("Transform1...")
    laptop.plug19()
  }
}

object Test {
  def main(args: Array[String]) {
    //you can do it like this:
    new LaptopAdapter(new Laptop).plug220()

    //or like this (doesn't need LaptopAdapter)
    new Laptop with Socket220V {
      def plug220() {
        println("Transform2...")
      }
    }
  }
}
```
program Adapter;

{$APPTYPE CONSOLE}
{$R *.res}

uses
  System.SysUtils;

type
  (* Its interface *)
  TTarget = class abstract
    function Request: string; virtual; abstract;
  end;

  (* A client accessed to this. *)
  TAdaptee = class(TTarget)
    function Request: string; override;
  end;

  (* Object Adapter uses composition and can wrap classes or interfaces, or both.*)
  (* Redirect call to Adaptee. It is loose coupling of client and adapter.*)
  (* It can do this since it contains, as a private, encapsulated member,
    the class or interface object instance it wraps. *)
  TObjectAdapter = class
    fAdaptee: TAdaptee;
    function SpecialRequest: string;
    constructor Create(adaptee: TAdaptee);
  end;

  constructor TObjectAdapter.Create;
  begin
    fAdaptee := TAdaptee.Create;
  end;

  function TObjectAdapter.SpecialRequest: string;
  begin
    Result := fAdaptee.Request;
  end;

  function TAdaptee.Request: string;
  begin
    Result := 'Adaptee';
  end;

var
  clientObject: TObjectAdapter;

begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
    clientObject := TObjectAdapter.Create(TAdaptee.Create);
Implementation

```delphi
WriteLn('Call method Object Adapter: '+clientObject.SpecialRequest);
WriteLn(#13#10+ 'Press any key to continue...');
ReadLn;
clientObject.Free;
except
  on E: Exception do
    WriteLn(E.ClassName, ': ', E.Message);
end;
end.
```

### 2.4.3 Class Adapter

```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())
```

```delphi
program Adapter;
{$APPTYPE CONSOLE}
{$R *.res}
uses
  System.SysUtils;
type
    (* Its interface *)
    TTarget = class abstract
      function Request: string; virtual; abstract;
    end;
    (* A client accessed to this. *)
    TAdaptee = class(TTarget)
      function Request: string; override;
    end;
    (* Class Adapter uses inheritance and can only wrap a class.*)
    (* This plain old inheritance. *)
    (* It cannot wrap an interface since by definition*)
    (* it must derive from some base class as Adaptee in example*)
    (*
```
* Can’t reuse Class Adapter without rewrite code
  * You need implements other adapter with other method in other class.
  */

TClassAdapter = class(TAdaptee)
  function SpecialRequest: string;
  end;

{ TClassAdapter }

function TClassAdapter.SpecialRequest: string;
begin
  //use inherited Request as SpecialRequest
  Result := inherited Request;
end;

{ TAdaptee }

function TAdaptee.Request: string;
begin
  Result := 'Adaptee';
end;

var
  clientClass: TClassAdapter;

begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
    clientClass := TClassAdapter.Create;
    WriteLn('Call method Class Adapter: ' + clientClass.SpecialRequest);
    WriteLn('#13#10 ' + 'Press any key to continue...');
    ReadLn;
    clientClass.Free;
  except
    on E: Exception do
      WriteLn(E.ClassName, ': ', E.Message);
  end;
end.
3 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:

|... |
| |... |
|... |

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.

| | |... |
|... | |... |
|... | |... |

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

### 3.1 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.

### 3.2 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.

#### 3.2.1 Creation

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

#### 3.2.2 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.
3.2.3 Removal

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

3.3 Advices

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

3.4 Implementation

```kotlin
/** "Implementor" */
fun interface MusicPlayer {
    fun play(song: String): String
}

/** "ConcreteImplementor" 1/4 */
val defaultPlayer = MusicPlayer{
    "Reproducing $it in $(MusicFormat.NONE)"
}

/** "ConcreteImplementor" 2/4 */
val mp3Player = MusicPlayer{
    "Reproducing $it in $(MusicFormat.MP3)"
}

/** "ConcreteImplementor" 3/4 */
val mp4Player = MusicPlayer{
    "Reproducing $it in $(MusicFormat.MP4)"
}

/** "ConcreteImplementor" 4/4 */
val vlcPlayer = MusicPlayer{
    "Reproducing "$it" in $(MusicFormat.VLC)"
}

abstract class MusicPlayerUI {
    var musicPlayer : MusicPlayer = defaultPlayer
    abstract fun playMusic(song: String, musicFormat: MusicFormat)
}

class MyMusicPlayerUI : MusicPlayerUI() {
    override fun playMusic(song: String, musicFormat: MusicFormat) {
        musicPlayer = when(musicFormat) {
            MusicFormat.NONE -> defaultPlayer
            MusicFormat.MP3 -> mp3Player
            MusicFormat.MP4 -> mp4Player
            MusicFormat.VLC -> vlcPlayer
        }
        println(musicPlayer.play(song))
    }
}

enum class MusicFormat{
    NONE, MP3, MP4, VLC
}
```
The following Java\(^1\) program illustrates the bridge pattern.

```java
/**
 * 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);
    }
}

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
```
Implementation
drawingAPI)
7 {
8 super(drawingAPI);
9 this.x = x;
10 this.y = y;
11 this.radius = radius;
12 }
13
14 // low-level i.e. Implementation specific
15 public void draw() {
16 drawingAPI.drawCircle(x, y, radius);
17 }
18
19 // high-level i.e. Abstraction specific
20 public void resizeByPercentage(double pct) {
21 radius *= pct;
22 }
23 }

/**
 * 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 7.500000
API2.circle at 5.000000:7.000000 radius 27.500000

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
3.4.1 C#

The following C#\textsuperscript{2} program illustrates the "shape" example given above and will output:

```csharp
using System;

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

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

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

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

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

/** "Client" */
class BridgePattern {
    public static void Main(string[] args) {
        IShape[] shapes = new IShape[2];
        shapes[0] = new CircleShape(x1, y1, r1, new DrawingAPI1());
        shapes[1] = new CircleShape(x2, y2, r2, new DrawingAPI2());
    }
}

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

\textsuperscript{2} https://en.wikibooks.org/wiki/C%20Sharp%20Programming
Bridge

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();
}

3.4.2 C# using generics

The following C#3 program illustrates the "shape" example given above and will output:

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

using System;

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

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

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

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

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

Implementation

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

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

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

The following Python4 program illustrates the "shape" example given above and will output:

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

# 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

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

# Refined Abstraction
class CircleShape(Shape):
    def __init__(self, x, y, radius, drawingAPI):

4  https://en.wikibooks.org/wiki/Python%20Programming
```
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()

An example in Ruby.\(^5\)

```ruby
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 necessary stuff'
  end
end

class ConcreteImplementorA < Implementor
  def operation
    super
    puts 'Doing additional stuff'
  end
end
```

A Scala\textsuperscript{6} implementation of the Java drawing example with the same output.

```scala
/** "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
}

/** "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)
}

6 https://en.wikibooks.org/wiki/Scala
// 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()
}

An example in D7.

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) {
    writeln("\nAPI1.circle at %f:%f radius %f", x, y, radius);
  }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2: DrawingAPI {
  void drawCircle(double x, double y, double radius) {
    writeln("\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;
  }

  // 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;
}

This example in Perl\(^8\) uses the MooseX::Declare\(^9\) module.

---

\(^8\) https://en.wikibooks.org/wiki/Perl%20Programming

\(^9\) http://search.cpan.org/search?query=MooseX%3A%3ADeclare&mode=all
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;

The following Delphi\textsuperscript{10} program illustrates the ”shape” example given above and will output:

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

program Bridge;
{$APPTYPE CONSOLE}
{$R *.res}
uses
System.SysUtils;
type

///implementator
TDrawingAPI = class abstract
    procedure DrawCircle(x, y, radius: double); virtual; abstract;
end;

//ConcreteImplementator 1
TDrawingAPI1 = class(TDrawingAPI)
    procedure DrawCircle(x, y, radius: double); override;
end;

//ConcreteImplementator 2
TDrawingAPI2 = class(TDrawingAPI)
    procedure DrawCircle(x, y, radius: double);override;
end;

//Abstraction
TShape = class abstract
    procedure Draw();virtual; abstract;// low-level (i.e. Implementation-specific)
    procedure ResizeByPercentage(pct: double);virtual; abstract;// high-level (i.e. Abstraction-specific)
end;

//Refined Abstraction
TCircleShape = class(TShape)
    strict private
        x, y, radius: double;
        drawingAPI: TDrawingAPI;
    public
        constructor Create(x, y, radius: double; drawingAPI: TDrawingAPI);
        procedure Draw;override;
        procedure ResizeByPercentage(pct: double);override;
    end;
end;

\textsuperscript{10} https://en.wikibooks.org/wiki/Delphi%20Programming
procedure TDrawingAPI1.DrawCircle(x, y, radius: double);
begin
  WriteLn('API1.circle at '+FloatToStr(x)+' : '+FloatToStr(y)+' radius '+FloatToStr(radius));
end;

{ TDeawingAPI }

procedure TDrawingAPI2.DrawCircle(x, y, radius: double);
begin
  WriteLn('API2.circle at '+FloatToStr(x)+' : '+FloatToStr(y)+' radius '+FloatToStr(radius));
end;

{ TCircleShape }

constructor TCircleShape.Create(x, y, radius: double; drawingAPI: TDrawingAPI);
begin
  self.x := x;
  self.y := y;
  self.radius := radius;
  self.drawingAPI := drawingAPI;
end;

procedure TCircleShape.Draw;
begin
  drawingAPI.DrawCircle(x, y, radius);
end;

procedure TCircleShape.ResizeByPercentage(pct: double);
begin
  radius := radius * pct;
end;

var shapes: array of TShape;
shape: TShape;
begin
  try
    { TODO -oUser -cConsole Main : Insert code here }
    SetLength(shapes, 2);
    shapes[0] := TCircleShape.Create(1, 2, 3, TDrawingAPI1.Create);
    shapes[1] := TCircleShape.Create(5, 7, 11, TDrawingAPI2.Create);

    for shape in shapes do
      begin
        shape.ResizeByPercentage(2.5);
        shape.Draw;
      end;

    WriteLn(#13#10+'Press any key to continue..');
    ReadLn;

    shapes[0].Free;
    shapes[1].Free;
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
The builder pattern is useful for avoiding a huge list of constructors for a class. Let's imagine a class that stores the mode of transport (of an employee for instance). Here is the constructor that takes a MetroLine object as a parameter:

```java
modeOfTransport(MetroLine)
```

Now we need a constructor for someone who uses the car, the bus or the train:

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

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

```java
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 who use several modes of transport to go to work. However, we realize that the list of constructors is now becoming a mess. Each new parameter leads to an exponential 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 by calling methods on it:

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

The list of methods is no longer exponential 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 single 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 in an UML class diagram:

![Builder Structure](image)

**Figure 13** 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 all at once. The best way to construct such an object is to use a builder class.
4.1 Examples

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

4.2 Cost

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

4.2.1 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 alongside your code. Then you will have to remove the existing constructor calls to use the builder instead. The refactoring is hardly automatic.

4.2.2 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 convert to an abstract factory.

4.2.3 Removal

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

4.3 Advices

- Put the `builder` term in the name of the builder class to indicate the use of the pattern to the other developers.
- 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.

4.4 Implementations

Building a car.

```java
/**
 * 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 {
```

---

1 Chapter 7 on page 101
2 Chapter 12 on page 159
/**
 * 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 build() {
        Car car = new Car(description);
        car.setCarType(carType);
        car.setHasTripComputer(hasTripComputer);
        car.setHasGPS(hasGPS);
        car.setSeaterNumber(seaterNumber);
        return car;
    }
}
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;
}
1 /**
2 * Construct a CarBuilder called carBuilder and build a car.
3 */
4 public class Director {
5    public static void main(String[] args) {
6        CarBuilder carBuilder = new CarBuilder();
7        carBuilder.setSeaters(2);
8        carBuilder.setSportCar();
9        carBuilder.setTripComputer();
10       carBuilder.unsetGPS();
11       Car car = carBuilder.getResult();
12       System.out.println(car);
13    }
14 }

It will produce:

A sport car with 2 seaters with a trip computer.

Building a pizza.

1 /** "Product" */
2 class Pizza {
3     private String dough = "";
4     private String sauce = "";
5     private String topping = "";
6     public void setDough(final String dough) {
7         this.dough = dough;
8     }
9     public void setSauce(final String sauce) {
10        this.sauce = sauce;
11     }
12     public void setTopping(final String topping) {
13        this.topping = topping;
14     }
15 }

1 /** "Abstract Builder" */
2 abstract class PizzaBuilder {
3     protected Pizza pizza;
4     public abstract void buildDough();
5     public abstract void buildSauce();
6     public abstract void buildTopping();
7     public void createNewPizzaProduct() {
8         this.pizza = new Pizza();
9     }
10    public Pizza getPizza() {
11        return this.pizza;
12    }
13 }

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");
    }
}

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");
    }
}

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();
    }
}

class BuilderExample {
    public static void main(final String[] args) {
        final Waiter waiter = new Waiter();

        final PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        final PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();
```csharp
waiter.setPizzaBuilder(hawaiianPizzaBuilder);
waiter.constructPizza();
final Pizza pizza = waiter.getPizza();
waiter.setPizzaBuilder(spicyPizzaBuilder);
waiter.constructPizza();
final Pizza anotherPizza = waiter.getPizza();
}
```
```csharp
private IBicycleBuilder _builder;

public MountainBikeBuildDirector(IBicycleBuilder builder)
{
    _builder = builder;
}

public void Construct()
{
    _builder.Colour = "Red";
    _builder.Height = 29;
}

public Bicycle GetResult()
{
    return this._builder.GetResult();
}

public class Client
{
    public void DoSomethingWithBicycles()
    {
        var director = new MountainBikeBuildDirector(new GTBuilder());
        // Director controls the stepwise creation of product and returns the
        // result.
        director.Construct();
        Bicycle myMountainBike = director.GetResult();
    }
}

Another example:

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()
        {
```
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
class Pizza
{
    public string dough = string.Empty;
This C++11 implementation is based on the pre C++98 implementation in the book.

```cpp
#include <iostream>

enum Direction {North, South, East, West};

class MapSite {
public:
    virtual void enter() = 0;
    virtual ~MapSite() = default;
};

class Room : public MapSite {
public:
    Room() : roomNumber(0) {}  // rule of three
    Room(int n) : roomNumber(n) {}  // rule of three
    void setSide(Direction d, MapSite* ms) {
        std::cout << "Room::setSide " << d << ' ' << ms << '\n';
    }
    virtual void enter() {}  // rule of three
    Room(const Room&) = delete;  // rule of three
    Room& operator=(const Room&) = delete;
private:
    int roomNumber;
};

class Wall : public MapSite {
public:
    Wall() {}
    virtual void enter() {}
};

class Door : public MapSite {
public:
    Door(Room* r1 = nullptr, Room* r2 = nullptr) : room1(r1), room2(r2) {}  // rule of three
    virtual void enter() {}
    Door(const Door&) = delete;  // rule of three
    Door& operator=(const Door&) = delete;
private:
    Room* room1;
    Room* room2;
};
```
Room* room1;
Room* room2;
};

class Maze {
public:
    void addRoom(Room* r) {
        std::cout << "Maze::addRoom " << r << \n";
    }
    Room* roomNo(int) const {
        return nullptr;
    }
};

class MazeBuilder {
public:
    virtual ~MazeBuilder() = default;
    virtual void buildMaze() = 0;
    virtual void buildRoom(int room) = 0;
    virtual void buildDoor(int roomFrom, int roomTo) = 0;
    virtual Maze* getMaze() {
        return nullptr;
    }
 protected:
    MazeBuilder() = default;
};

// If createMaze is passed an object that can create a new maze in its entirety
using operations for adding rooms, doors, and walls to the maze it builds, then
you can use inheritance to change parts of the maze or the way the maze is
built. This is an example of the Builder (110) pattern.

class MazeGame {
public:
    Maze* createMaze(MazeBuilder& builder) {
        builder.buildMaze();
        builder.buildRoom(1);
        builder.buildRoom(2);
        builder.buildDoor(1, 2);
        return builder.getMaze();
    }
    Maze* CreateComplexMaze(MazeBuilder& builder) {
        builder.buildRoom(1);
        // ...
        builder.buildRoom(1001);
        return builder.getMaze();
    }
};
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder() : currentMaze(nullptr) {}
    virtual void buildMaze() {
        currentMaze = new Maze;
    }
    virtual void buildRoom(int n) {
        if (!currentMaze->roomNo(n)) {
            Room* room = new Room(n);
            currentMaze->addRoom(room);
            room->setSide(North, new Wall);
            room->setSide(South, new Wall);
            room->setSide(East, new Wall);
            room->setSide(West, new Wall);
        }
    }
};
Implementations

```cpp
template<>
virtual void buildDoor(int n1, int n2) {
    Room* r1 = currentMaze->roomNo(n1);
    Room* r2 = currentMaze->roomNo(n2);
    Door* d = new Door(r1, r2);
    r1->setSide(commonWall(r1, r2), d);
    r2->setSide(commonWall(r2, r1), d);
}

virtual Maze* getMaze() {
    return currentMaze;
}

StandardMazeBuilder(const StandardMazeBuilder&) = delete; // rule of three
StandardMazeBuilder& operator=(const StandardMazeBuilder&) = delete;

private:
    Direction commonWall(Room*, Room*) {
        return North;
    }
    Maze* currentMaze;
};
```

```cpp
int main() {
    MazeGame game;
    StandardMazeBuilder builder;
    game.createMaze(builder);
    builder.getMaze();
}
```

The program output is like:

```
Maze::addRoom 0x2145ed0
Room::setSide 0 0x2146300
Room::setSide 1 0x2146320
Room::setSide 2 0x2146340
Room::setSide 3 0x2146360
Maze::addRoom 0x2146380
Room::setSide 0 0x21463a0
Room::setSide 1 0x21463c0
Room::setSide 2 0x21463e0
Room::setSide 3 0x2146400
Room::setSide 0 0x2146420
Room::setSide 0 0x2146420
```

Another example:

```cpp
#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, " <<
```

57
sausage 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.
```cpp
{  
  ~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();  
}
```
echo "Sauce : 
".$this->sauce."<br/>
";  

```
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();
}
```

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

```
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");
    }
}
```

```
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");
    }
}
```

```
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

# 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
Implementations

class Animal:
    ***
    Abstract Animal
    ***
    legs = 0
tail = False
roar = ''

class Mutator:
    ***
    Mutator
    ***
    def mutate(self):
        self.animal = Animal()

class Cat(Mutator):
    ***
    Cat
    ***
    def create_legs(self):
        self.animal.legs = 4
    def create_tail(self):
        self.animal.tail = True
    def create_roar(self):
        self.animal.roar = 'meowww!'

class Dog(Mutator):
    ***
    Dog
    ***
    def create_legs(self):
        self.animal.legs = 4
    def create_tail(self):
        self.animal.tail = True
    def create_roar(self):
        self.animal.roar = 'woffff!'  

class AnimalOwner:
    ***
    Farm owner
    ***
    __mutator = ''
    def set_animal(self, mutator):
        self.__mutator = mutator
    def create_an_animal_for_me(self):
        self.__mutator.mutate()
        self.__mutator.create_legs()
        self.__mutator.create_tail()
        self.__mutator.create_roar()
    def get_animal(self):
        return self.__mutator.animal

c = Cat()
d = Dog()
ao = AnimalOwner()
ao.set_animal(c)
ao.create_an_animal_for_me()
animal = ao.get_animal()
print(animal.roar)  # meowww!
program Builder;

{$APPTYPE CONSOLE}

{$R *.res}

uses
  System.SysUtils;

type

  {Product - final object that will be created by Director using Builder}*
  TPizza = class
    fDough: string;
    fSauce: string;
    fTopping: string;
    procedure Open;
  end;

  {Builder - abstract class as interface for creating objects -> product in this case}*
  TPizzaBuilder = class abstract
    protected
      fPizza: TPizza;
    public
      function GetPizza: TPizza;
      procedure CreateNewPizzaProduct;
      procedure BuildDough; virtual; abstract;
      procedure BuildSauce; virtual; abstract;
      procedure BuildTopping; virtual; abstract;
  end;

  {concrete builder - provides implementation for Builder}*
  (an object able to construct other objects.*)
  THawaiianPizzaBuilder = class(TPizzaBuilder)
    public
      procedure BuildDough; override;
      procedure BuildSauce; override;
      procedure BuildTopping; override;

  end;

  {concrete builder - provides implementation for Builder}*
  (an object able to construct other objects.*)
  TSpicyPizzaBuilder = class(TPizzaBuilder)
    public
      procedure BuildDough; override;
      procedure BuildSauce; override;
      procedure BuildTopping; override;

  end;

  {Director - responsible for managing the correct sequence of object creation.}
  (Receives a Concrete Builder as a parameter and executes the necessary operations on it.)*
  TCook = class
    protected
      fPizzaBuilder: TPizzaBuilder;
    public
      procedure SetPizzaBuilder(pb: TPizzaBuilder);
      function GetPizza: TPizza;
      procedure ConstructPizza;
  end;

{ THawaiianPizzaBuilder }

procedure THawaiianPizzaBuilder.BuildDough;
begin
fPizza.fDough:= 'cross';
end;

procedure THawaiianPizzaBuilder.BuildSauce;
begin
  fPizza.fSauce:= 'mild';
end;

procedure THawaiianPizzaBuilder.BuildTopping;
begin
  fPizza.fTopping:= 'ham + pineapple';
end;

{ TSpicyPizzaBuilder }
procedure TSpicyPizzaBuilder.BuildDough;
begin
  fPizza.fDough:= 'pan baked';
end;

procedure TSpicyPizzaBuilder.BuildSauce;
begin
  fPizza.fSauce:= 'hot';
end;

procedure TSpicyPizzaBuilder.BuildTopping;
begin
  fPizza.fTopping:= 'pepperoni + salami';
end;

{ TCook }
procedure TCook.ConstructPizza;
begin
  fPizzaBuilder.CreateNewPizzaProduct;
  fPizzaBuilder.BuildDough;
  fPizzaBuilder.BuildSauce;
  fPizzaBuilder.BuildTopping;
end;

function TCook.GetPizza: TPizza;
begin
  Result:= fPizzaBuilder.GetPizza;
end;

procedure TCook.SetPizzaBuilder(pb: TPizzaBuilder);
begin
  fPizzaBuilder:= pb;
end;

{ TPizzaBuilder }
procedure TPizzaBuilder.CreateNewPizzaProduct;
begin
  fPizza:= TPizza.Create;
end;

function TPizzaBuilder.GetPizza: TPizza;
begin
  Result:= fPizza;
end;

{ TPizza }
procedure TPizza.Open;
begin
WriteLn('Pizza with: ' + fDough + ' dough, ' + fSauce + ' sauce and ' + fTopping + ' topping. Mmm.');
end;

var
cook: TCook;
hawaiian, spicy: TPizza;

begin
  try
    cook:= TCook.Create;
    cook.SetPizzaBuilder(THawaiianPizzaBuilder.Create);
    cook.ConstructPizza;
    WriteLn('(* create the product *)');
    hawaiian:= cook.GetPizza;
    hawaiian.Open;
    WriteLn('#13#10'+'(* create another product *)');
    cook.SetPizzaBuilder(TSpicyPizzaBuilder.Create);
    cook.ConstructPizza;
    spicy:= cook.GetPizza;
    spicy.Open;
    WriteLn('#13#10'+'Press any key to continue...');
    ReadLn;
    spicy.Free;
    hawaiian.Free;
    cook.Free;
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
5 Chain of responsibility

Various examples of the Chain of responsibility pattern.

Below is an example of this pattern in Java. A logger is created using a chain of loggers, each one configured with different log levels.
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.
if (next != null) {
    next.message(msg, priority);
}

abstract protected void writeMessage(String msg);

class StdoutLogger extends Logger {
    public StdoutLogger(int mask) {
        super(mask);
    }
    protected void writeMessage(String msg) {
        System.out.println("Writing to stdout: "+msg);
    }
}

class EmailLogger extends Logger {
    public EmailLogger(int mask) {
        super(mask);
    }
    protected void writeMessage(String msg) {
        System.out.println("Sending via email: "+msg);
    }
}

class StderrLogger extends Logger {
    public StderrLogger(int mask) {
        super(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:

```plaintext
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`.

```java
abstract class PurchasePower {
    protected double final 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

```java
class ManagerPPower extends PurchasePower {
    private double final 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);
        }
    }
}
```

```java
class DirectorPPower extends PurchasePower {
    private double final ALLOWABLE = 20 * base;

    public void processRequest(PurchaseRequest request) {
        if(request.getAmount() < ALLOWABLE) {
            System.out.println("Director will approve $" + request.getAmount());
        } else if(successor != null) {
```
Implementations

```java
successor.processRequest(request);
}
}
}

class VicePresidentPPower extends PurchasePower {
    private double final 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 double final 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.

```java
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);
        }
    }
}

"""Chain of responsibility pattern example."""
from abc import ABCMeta, abstractmethod
from enum import Enum, auto

class LogLevel(Enum):
    """Log Levels Enum."""
    NONE = auto()
    INFO = auto()
    DEBUG = auto()
    WARNING = auto()
    ERROR = auto()
    FUNCTIONAL_MESSAGE = auto()
    FUNCTIONAL_ERROR = auto()
    ALL = auto()

class Logger:
    """Abstract handler in chain of responsibility pattern."""

72
```python
__metaclass__ = ABCMeta

next = None

def __init__(self, levels) -> None:
    """Initialize new logger.
    Arguments:
    levels (list[str]): List of log levels.
    """
    self.log_levels = []
    for level in levels:
        self.log_levels.append(level)

def set_next(self, next_logger: Logger):
    """Set next responsible logger in the chain.
    Arguments:
    next_logger (Logger): Next responsible logger.
    Returns: Logger: Next responsible logger.
    """
    self.next = next_logger
    return self.next

def message(self, msg: str, severity: LogLevel) -> None:
    """Message writer handler.
    Arguments:
    msg (str): Message string.
    severity (LogLevel): Severity of message as log level enum.
    """
    if LogLevel.ALL in self.log_levels or severity in self.log_levels:
        self.write_message(msg)
    if self.next is not None:
        self.next.message(msg, severity)

@abstractmethod
def write_message(self, msg: str) -> None:
    """Abstract method to write a message.
    Arguments:
    msg (str): Message string.
    Raises: NotImplementedError
    """
    raise NotImplementedError("You should implement this method.")

class ConsoleLogger(Logger):
    def write_message(self, msg: str) -> None:
        """Overrides parent's abstract method to write to console.
        Arguments:
        msg (str): Message string.
        """
        print("Writing to console:", msg)

class EmailLogger(Logger):
    def write_message(self, msg: str) -> None:
        """Overrides parent's abstract method to send an email.
        Arguments:
        msg (str): Message string.
        """
        print(f"Sending via email: {msg}")
```

class FileLogger(Logger):
    def write_message(self, msg: str) -> None:
        """Overides parent's abstract method to write a file.
        Arguments:
        msg (str): Message string.
        """
        print(f"Writing to log file: {msg}\n")

def main():
    """Building the chain of responsibility.\n    """
    logger = ConsoleLogger([LogLevel.ALL])
    email_logger = logger.set_next(EmailLogger([LogLevel.FUNCTIONAL_MESSAGE, LogLevel.FUNCTIONAL_ERROR]))
    # As we don't need to use file logger instance anywhere later
    # We will not set any value for it.
    email_logger.set_next(FileLogger([LogLevel.WARNING, LogLevel.ERROR]))

    # ConsoleLogger will handle this part of code since the message
    # has a log level of all
    logger.message("Entering function ProcessOrder().", LogLevel.DEBUG)
    logger.message("Order record retrieved.\n", LogLevel.INFO)

    # ConsoleLogger and FileLogger will handle this part since file logger
    # implements WARNING and ERROR
    logger.message("Customer Address details missing in Branch DataBase." , LogLevel.WARNING)
    logger.message("Customer Address details missing in Organization DataBase.\n", LogLevel.ERROR)

    # ConsoleLogger and EmailLogger will handle this part as they implement
    # functional error
    logger.message("Unable to Process Order ORD1 Dated D1 for customer C1." , LogLevel.FUNCTIONAL_ERROR)
    logger.message("OrderDispatched." , LogLevel.FUNCTIONAL_MESSAGE)

if __name__ == "__main__":
    main()

Another example:

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())

e num LogLevel
    None
    Info
    Debug
    Warning
    Error
    FunctionalMessage
    FunctionalError
    All
end

abstract class Logger
    property log_levels
    property next : Logger | Nil

    def initialize(*levels)
        @log_levels = [] of LogLevel

        levels.each do |level|
            @log_levels << level
        end

    end

    def message(msg : String, severity : LogLevel)
        if @log_levels.includes?(LogLevel::All) || @log_levels.includes?(severity)
            write_message(msg)
        end
        @next.try(&.message(msg, severity))
    end

    abstract def write_message(msg : String)
end

class ConsoleLogger < Logger
    def write_message(msg : String)
        puts "Writing to console: #{msg}"
    end
class EmailLogger < Logger
def write_message(msg : String)
  puts "Sending via email: #{msg}"
end
end
class FileLogger < Logger
def write_message(msg : String)
  puts "Writing to Log File: #{msg}"
end
end

# Program
# Build the chain of responsibility
logger = ConsoleLogger.new(LogLevel::All)
logger1 = logger.next = EmailLogger.new(LogLevel::FunctionalMessage, LogLevel::FunctionalError)
logger2 = logger1.next = FileLogger.new(LogLevel::Warning, LogLevel::Error)

# Handled by ConsoleLogger since the console has a loglevel of all
logger.message("Entering function ProcessOrder()").
logger.message("Order record retrieved.", LogLevel::Info)

# Handled by ConsoleLogger and FileLogger since filelogger implements Warning & Error
logger.message("Customer Address details missing in Branch DataBase.", LogLevel::Warning)
logger.message("Customer Address details missing in Organization DataBase.", LogLevel::Error)

# Handled by ConsoleLogger and EmailLogger as it implements functional error
logger.message("Unable to Process Order ORD1 Dated D1 For Customer C1.", LogLevel::FunctionalError)

# Handled by ConsoleLogger and EmailLogger
logger.message("Order Dispatched.", LogLevel::FunctionalMessage)
public const WARNING = 0b000100;
public const ERROR = 0b001000;
public const FUNCTIONAL_MESSAGE = 0b010000;
public const FUNCTIONAL_ERROR = 0b100000;
public const ALL = 0b111111;

/** @var int A bitmask flag from this class. */
protected int $logMask;

/** @var \Logger|null An optional next logger to handle the message */
protected ?Logger $next = null;

/**
 * Logger constructor.
 *
 * @param int $mask
 * A bitmask flag from this class.
 */
public function __construct(int $mask)
{
    $this->logMask = $mask;
}

/**
 * Set next responsible logger in the chain.
 *
 * @param \Logger $nextLogger
 * Next responsible logger.
 *
 * @return \Logger
 * Logger: Next responsible logger.
 */
public function setNext(Logger $nextLogger): Logger
{
    $this->next = $nextLogger;
    return $nextLogger;
}

/**
 * Message writer handler.
 *
 * @param string $msg
 * Message string.
 *
 * @param int $severity
 * Severity of message as a bitmask flag from this class.
 *
 * @return $this
 */
public function message(string $msg, int $severity): Logger
{
    if ($severity & $this->logMask) {
        $this->writeMessage($msg);
    }
    if ($this->next !== null) {
        $this->next->message($msg, $severity);
    }

    return $this;
}

/**
 * Abstract method to write a message
 *
 * @param string $msg
 * Message string.
 */
abstract protected function writeMessage(string $msg): void;
}

class ConsoleLogger extends Logger
{
    protected function writeMessage(string $msg): void
    {
        echo "Writing to console: $msg\n";
    }
}

class EmailLogger extends Logger
{
    protected function writeMessage(string $msg): void
    {
        echo "Sending via email: $msg\n";
    }
}

class FileLogger extends Logger
{
    protected function writeMessage(string $msg): void
    {
        echo "Writing to a log file: $msg\n";
    }
}

$logger = new ConsoleLogger(Logger::ALL);
$logger->setNext(new EmailLogger(Logger::FUNCTIONAL_MESSAGE | Logger::FUNCTIONAL_ERROR))->setNext(new FileLogger(Logger::WARNING | Logger::ERROR));

$logger
    // Handled by ConsoleLogger since the console has a loglevel of all
    ->message("Entering function ProcessOrder()", Logger::DEBUG)
    ->message("Order record retrieved.", Logger::INFO)
    // Handled by ConsoleLogger and FileLogger since filelogger implements Warning & Error
    ->message("Customer Address details missing in Branch Database.", Logger::WARNING)
    ->message("Customer Address details missing in Organization Database.", Logger::ERROR)
    // Handled by ConsoleLogger and EmailLogger as it implements functional error
    ->message("Unable to Process Order ORD1 Dated D1 For Customer C1.", Logger::FUNCTIONAL_ERROR)
    // Handled by ConsoleLogger and EmailLogger
    ->message("Order Dispatched.", Logger::FUNCTIONAL_MESSAGE);

/* Output
Writing to console: Entering function ProcessOrder().
Writing to console: Order record retrieved.
Writing to a log file: Customer Address details missing in Branch Database.
Writing to a log file: Customer Address details missing in Organization Database.
Writing to a log file: Customer Address details missing in Organization Database.
*/
Implementations

Writing to console: Unable to Process Order ORD1 Dated D1 For Customer C1.
Sending via email: Unable to Process Order ORD1 Dated D1 For Customer C1.
Writing to console: Order Dispatched.
Sending via email: Order Dispatched.
*/

#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

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 => "rv", 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);
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 );

This C# examples uses the logger application to select different sources based on the log level;

namespace ChainOfResponsibility;

[Flags]
public enum LogLevel
{
    None = 0,    /**< 0 */
    Info = 1,    /**< 1 */
};
public abstract class Logger
{
    protected LogLevel logMask;

    // The next Handler in the chain
    protected Logger next;

    public Logger(LogLevel mask)
    {
        this.logMask = mask;
    }

    public Logger SetNext(Logger nextlogger)
    {
        Logger lastLogger = this;
        while (lastLogger.next != null)
        {
            lastLogger = lastLogger.next;
        }
        lastLogger.next = nextlogger;
        return this;
    }

    public void Message(string msg, LogLevel severity)
    {
        if ((severity & logMask) != 0) // True only if any of the logMask bits are set in severity
        {
            WriteMessage(msg);
        }
        if (next != null)
        {
            next.Message(msg, severity);
        }
    }

    abstract protected void WriteMessage(string msg);
}

public class ConsoleLogger : Logger
{
    public ConsoleLogger(LogLevel mask)
    :
        base(mask)
    {
    }

    protected override void WriteMessage(string msg)
    {
        Console.WriteLine("Writing to console: "+ msg);
    }
}
public class EmailLogger : Logger
{
    public EmailLogger(LogLevel mask)
        : base(mask)
    {
    
        protected override void WriteMessage(string msg)
        {
            // Placeholder for mail send logic, usually the email configurations are
            // saved in config file.
            Console.WriteLine("Sending via email: " + msg);
        }
    }
}

class FileLogger : Logger
{
    public FileLogger(LogLevel mask)
        : base(mask)
    {
    
        protected override void WriteMessage(string msg)
        {
            // Placeholder for File writing logic
            Console.WriteLine("Writing to Log File: " + msg);
        }
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Build the chain of responsibility
        Logger logger;
        logger = new ConsoleLogger(LogLevel.All)
            .SetNext(new EmailLogger(LogLevel.FunctionalMessage | LogLevel.FunctionalError))
            .SetNext(new FileLogger(LogLevel.Warning | LogLevel.Error));

        // Handled by ConsoleLogger since the console has a loglevel of all
        logger.Message("Entering function ProcessOrder().", LogLevel.Debug);
        logger.Message("Order record retrieved.", LogLevel.Info);

        // Handled by ConsoleLogger and FileLogger since filelogger implements
        Warning & Error
        logger.Message("Customer Address details missing in Branch DataBase.",
            LogLevel.Warning);
        logger.Message("Customer Address details missing in Organization
        DataBase.", LogLevel.Error);

        // Handled by ConsoleLogger and EmailLogger as it implements functional
        error
        logger.Message("Unable to Process Order ORD1 Dated D1 For Customer C1.",
            LogLevel.FunctionalError);

        // Handled by ConsoleLogger and EmailLogger
        logger.Message("Order Dispatched.", LogLevel.FunctionalMessage);
    }
}

/* Output
Writing to console: Entering function ProcessOrder().
Writing to console: Order record retrieved.
Writing to Log File: Customer Address details missing in Branch DataBase.
Writing to console: Customer Address details missing in Branch DataBase.
Writing to console: Customer Address details missing in Organization DataBase.
*/
Another example:

```csharp
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();
    }
}
```

```csharp
// "Handler"
abstract class Handler
{
    protected Handler successor;

    public void SetSuccessor(Handler successor)
    {
        this.successor = successor;
    }

    public abstract void HandleRequest(int request);
}
```

```csharp
// "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);
        }
    }
}
Implementations

// "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);
        }
    }
}
6 Command

The command pattern is an object behavioural pattern that decouples sender and receiver. It can also be thought as an object-oriented equivalent of a call back method. Call back: It is a function that is registered to be called at a later point of time based on user actions.

6.1 Scope

Object

6.2 Purpose

Behavioral

6.3 Intent

Encapsulate the request for a service as an object.

6.4 Applicability

- to parameterize objects with an action to perform
- to specify, queue, and execute requests at different times
- for a history of requests
- for multilevel undo/redo

6.5 Structure

Figure 14  UML diagram of the command pattern
6.6 Consequences

- abstracts executor of a service
- supports arbitrary-level undo-redo
- composition yields macro-commands
- might result in lots of trivial command subclasses

6.7 Examples

Figure 15

The best example for this pattern is the graphical user interface. On most multidata interfaces, you have both a "New" menu and a "New" button with an icon like in Libre Office Writer. Both controls are connected to a command object, so the action is performed by the same code.

6.8 Cost

This pattern is dealing with the whole architecture of a program. It may have a substantial impact on the project.
6.8.1 Creation
If the code already exists, this pattern is very expensive.

6.8.2 Maintenance
This pattern is very expensive to maintain.

6.8.3 Removal
This pattern is very expensive to remove, too.

6.9 Advises

• Use the Command term to indicate the use of the pattern to the other developers.

6.10 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.

```java
/* The Command interface */
public interface Command {
    void execute();
}

import java.util.List;
import java.util.ArrayList;
/*@ The Invoker class */
public class Switch {
    private List<Command> history = new ArrayList<Command>();
    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){
        // Check number of arguments
        if (args.length != 1) {
            System.err.println("Argument "\"ON\" or "\"OFF\" is required.");
            System.exit(-1);
        }
        Light lamp = new Light();
        Command switchUp = new FlipUpCommand(lamp);
        Command switchDown = new FlipDownCommand(lamp);
        // See criticism of this model above:
        // The switch itself should not be aware of lamp details (switchUp,
        // switchDown)
        // either directly or indirectly
        Switch mySwitch = new Switch();
        switch (args[0]) {
            case "ON":
                mySwitch.storeAndExecute(switchUp);
                break;
            case "OFF":
                mySwitch.storeAndExecute(switchDown);
                break;
            default:
                System.err.println("Argument "\"ON\" or "\"OFF\" is required.");
                System.exit(-1);
        }
    }
}

6.11 Operations
The implementations to do are:
• copying a command before putting it on a history list
• handling hysteresis
• supporting transactions

```java
public interface Command {
    public int execute(int a, int b);
}
```

```java
public class AddCommand implements Command {
    public int execute(int a, int b) {
        return a + b;
    }
}
```

```java
public class MultCommand implements Command {
    public int execute(int a, int b) {
        return a * b;
    }
}
```

```java
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.

### 6.11.1 Command in Java

Menus and buttons provide a classic example of the need for the command pattern. When you add a menu to an application, you have to configure the menu with words that describe actions that the user can choose, such as Save and Open. Similarly for a button. You also have to configure the menu or button so that it can take action, calling a method in response to a user's click. However, JMenuItem or JButton class has no way of knowing what action to perform when an item or button is selected. In the following example we use the same Action for both a menu item and a button saving us from having to write the same code twice.

```java
Action actionOpen = new AbstractAction("Open...", iconOpen) {
    public void actionPerformed(ActionEvent e) {
        ... // open a file
    }
}
```

```java
JMenu mFile = new JMenu("File");
JMenuItem item = mFile.add(actionOpen); // use the same action for both a
menu item ...
JToolBar m_toolBar = new JToolBar();
JButton bOpen = new JButton(actionOpen); // ... and a button
m_toolBar.add(bOpen);
```

Java Swing applications commonly apply the Mediator pattern, registering a single object to receive all GUI events. This object mediates the interaction of the components and translates user input into commands for business domain objects.
6.11.2 Undo in Java

Swing provides for Undo/Redo functionality.

```java
import javax.swing.undo.*;

UndoManager undoManager = new UndoManager();
Action undoAction, redoAction;
undoAction = new AbstractAction("Undo",
    new ImageIcon("edit_undo.gif")) {
    public void actionPerformed(ActionEvent e) {
        try {
            undoManager.undo(); //undoManager.redo();
        }
        catch (CannotUndoException ex) {
            System.err.println("Unable to undo: "+ex);
        }
        updateUndo();
    }
    protected void updateUndo() {
        if(undo.canUndo()) {
            undoAction.setEnabled(true);
            undoAction.putValue(Action.NAME, undo.getUndoPresentationName());
        } else {
            undoAction.setEnabled(false);
            undoAction.putValue(Action.NAME, "Undo");
        }
        if(undo.canRedo()) {
            redoAction.setEnabled(true);
            redoAction.putValue(Action.NAME, undo.getRedoPresentationName());
        } else {
            redoAction.setEnabled(false);
            redoAction.putValue(Action.NAME, "Redo");
        }
    }
}
```

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 C# implementation 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.

```csharp
using System;

namespace CommandPattern{
public interface ICommand{
    void Execute();
}
/* The Invoker class */
public class Switch{
    ICommand _closedCommand;
    ICommand _openedCommand;
    public Switch(ICommand closedCommand, ICommand openedCommand)
}```
```csharp
// Close the circuit / power on
public void Close()
{
    _closedCommand.Execute();
}

// Open the circuit / power off
public void Open()
{
    _openedCommand.Execute();
}

/* An interface that defines actions that the receiver can perform */
public interface ISwitchable
{
    void PowerOn();
    void PowerOff();
}

/*/ The Receiver class */
public class Light : ISwitchable
{
    public void PowerOn()
    {
        Console.WriteLine("The light is on");
    }
    public void PowerOff()
    {
        Console.WriteLine("The light is off");
    }
}

/*/ The Command for turning off the device - ConcreteCommand #1 */
public class CloseSwitchCommand : ICommand
{
    private ISwitchable _switchable;
    public CloseSwitchCommand(ISwitchable switchable)
    {
        _switchable = switchable;
    }
    public void Execute()
    {
        _switchable.PowerOff();
    }
}

/*/ The Command for turning on the device - ConcreteCommand #2 */
public class OpenSwitchCommand : ICommand
{
    private ISwitchable _switchable;
    public OpenSwitchCommand(ISwitchable switchable)
    {
        _switchable = switchable;
    }
    public void Execute()
    {
    }
}
```
public interface ICommand
{
    void Execute();
}

/* The Invoker class */
public class Switch
{
    private List<ICommand> _commands = new List<ICommand>();
    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");
    }
}
```csharp
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.");
        }
    }
}
```

The following code is an implementation of command pattern in Python.

```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("****")

/* 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)
The following code is an implementation of command pattern in Javascript.

```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();
```

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 |
lamp := Light new.
switchUp := Message receiver: lamp selector: #turnOn.
switchDown := Message receiver: lamp selector: #turnOff.
state = #on ifTrue: [
^switch flipUp].
state = #off ifTrue: [
^switch flipDown].
Transcript show: 'Argument #on or #off is required.'.
!!

The following code is an implementation of command pattern in PHP.

```php
interface Command
{
    public function execute();
}

/** The Invoker class */
class Switcher
{
    private $history = array();

    public function storeAndExecute(Command $cmd)
    {
        $this->history[] = $cmd;
        $cmd->execute();
    }
}

/** The Receiver class */
class Light
{
    public function turnOn()
    {
        echo("The light is on");
    }

    public function turnOff()
    {
        echo("The light is off");
    }
```
/** The Command for turning on the light - ConcreteCommand #1 */
class FlipUpCommand implements Command {
    private $theLight;

    public function __construct(Light $light) {
        $this->theLight = $light;
    }

    public function execute() {
        $this->theLight->turnOn();
    }
}

/** The Command for turning off the light - ConcreteCommand #2 */
class FlipDownCommand implements Command {
    private $theLight;

    public function __construct(Light $light) {
        $this->theLight = $light;
    }

    public function execute() {
        $this->theLight->turnOff();
    }
}

class PressSwitch {
    public static function main(string $args) {
        $lamp = new Light();
        $switchUp = new FlipUpCommand($lamp);
        $switchDown = new FlipDownCommand($lamp);

        $mySwitch = new Switcher();

        switch($args) {
            case 'ON':
                $mySwitch->storeAndExecute($switchUp);
                break;
            case 'OFF':
                $mySwitch->storeAndExecute($switchDown);
                break;
            default:
                echo("Argument \"ON\" or \"OFF\" is required.");
                break;
        }
    }
}

/*INPUT*/
PressSwitch::main('ON'). PHP_EOL;
PressSwitch::main('OFF'). PHP_EOL;

/*OUTPUT*/
//The light is on
//The light is off
7 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 leaves separately. All of this creates a big amount of code. Suppose that you have to handle a file system repository. 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:

```pseudo
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 16  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:

```java
interface FileSystemComponent {
    method searchFilesInFolders(searchedFileName) is
        input: the searchedFileName that should be found in the folders.
        output: the list of encountered files.
```
As you can see, a component can be an individual object and also can be a collection of objects. A Composite pattern can represent both the conditions. In this pattern, one can develop tree structures for representing part-whole hierarchies.

### 7.1 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.

### 7.2 Cost

This pattern is one of the least 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.

#### 7.2.1 Creation

You have to create an almost empty interface and implement the management of the composite children. This cost is very low.
7.2.2 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.

7.2.3 Removal

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

7.3 Advices

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

7.4 Implementations

Various examples of the composite pattern.

```csharp
using System;
using System.Collections.Generic;
namespace Composite
{
    class Program
    {
        interface IGraphic
        {
            void Print();
        }
        class CompositeGraphic : IGraphic
        {
            private List<IGraphic> child = new List<IGraphic>();
            public CompositeGraphic(IEnumerable<IGraphic> 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()
                })
            })
        }
    }
}```
The following example, written in Common Lisp\(^1\), 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`.

\[
\text{(defstruct ellipse) ;; An empty struct.}
\]

\[
\text{;; For the method definitions, "object" is the variable,}
\]

\[
\text{;; and the following word is the type.}
\]

\[
\text{(defmethod print-graphic ((object null))}
\]

\[
\text{NIL)}
\]

\[
\text{(defmethod print-graphic ((object cons))}
\]

\[
\text{(print-graphic (first object))}
\]

\[
\text{(print-graphic (rest object))}
\]

\[
\text{(defmethod print-graphic ((object ellipse))}
\]

\[
\text{(print 'ELLIPSE))}
\]

\[
\text{(let* ((ellipse-1 (make-ellipse))}
\]

\[
\text{(ellipse-2 (make-ellipse))}
\]

\[
\text{(ellipse-3 (make-ellipse))}
\]

\[
\text{(ellipse-4 (make-ellipse))}
\]

\[
\text{(print-graphic (cons (list ellipse-1 (list ellipse-2 ellipse-3)) ellipse-4)))}
\]

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

\[
\text{Graphic ::= ellipse | GraphicList}
\]

\[
\text{GraphicList ::= empty | Graphic GraphicList}
\]

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

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

/** "Component" */
interface Graphic {
    //Prints the graphic.
    public void print();
}

/** "Composite" */
class CompositeGraphic implements Graphic {
    //Collection of child graphics.
    private final List<Graphic> childGraphics = new ArrayList<>();

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

---

2. [https://en.wikibooks.org/wiki/translation%20%28geometry%29](https://en.wikibooks.org/wiki/translation%20%28geometry%29)
The following example, written in Java\(^3\), implements a graphic class, which can be either an ellipse or a composition of several graphics. Every graphic can be printed. In algebraic form,

\[
\text{Graphic} = \text{ellipse} \mid \text{GraphicList} \\
\text{GraphicList} = \text{empty} \mid \text{Graphic} \text{ GraphicList}
\]

It could be extended to implement several other shapes (rectangle, etc.) and methods (translate\(^4\), etc.).

---

3 [https://en.wikibooks.org/wiki/Subject%3AJava%20programming%20language](https://en.wikibooks.org/wiki/Subject%3AJava%20programming%20language)

1 /** "Component" */
2 interface Graphic {
3     // Prints the graphic.
4     public void print();
5 }

1 /** "Composite" */
2 import java.util.List;
3 import java.util.ArrayList;
4 class CompositeGraphic implements Graphic {
5     // Collection of child graphics.
6     private List<Graphic> mChildGraphics = new ArrayList<Graphic>();
7     // Prints the graphic.
8     public void print() {
9         // Adds the graphic to the composition.
10        mChildGraphics.add(graphic);
11     }
12     // Removes the graphic from the composition.
13     public void remove(Graphic graphic) {
14         mChildGraphics.remove(graphic);
15     }
16     // Adds the graphic to the composition.
17     public void add(Graphic graphic) {
18         mChildGraphics.add(graphic);
19     }
20     // Prints the graphic.
21     public void print() {
22         for (Graphic graphic : mChildGraphics) {
23             graphic.print();
24         }
25     }
26     // Prints the complete graphic (four times the string "Ellipse").
27     public void print() {
28         // Adds the graphic to the composition.
29         mChildGraphics.add(graphic);
30         // Removes the graphic from the composition.
31         mChildGraphics.remove(graphic);
32     }
33 }

1 /** "Leaf" */
2 class Ellipse implements Graphic {
3     // Prints the graphic.
4     public void print() {
5         System.out.println("Ellipse");
6     }
7 }

1 /** Client */
2 public class Program {
3     public static void main(String[] args) {
4         // Initialize four ellipses
5         Ellipse ellipse1 = new Ellipse();
6         Ellipse ellipse2 = new Ellipse();
7         Ellipse ellipse3 = new Ellipse();
8         Ellipse ellipse4 = new Ellipse();
9         // Initialize three composite graphics
10        CompositeGraphic graphic = new CompositeGraphic();
11        CompositeGraphic graphic1 = new CompositeGraphic();
12        CompositeGraphic graphic2 = new CompositeGraphic();
13         // Composes the graphics
14        graphic1.add(ellipse1);
15        graphic1.add(ellipse2);
16        graphic1.add(ellipse3);
17        graphic2.add(ellipse4);
18        graphic.add(graphic1);
19        graphic.add(graphic2);
20         // Prints the complete graphic (four times the string "Ellipse").
21         graphic.print();
22     }
23 }

<?php
1 /** "Component" */
2 interface Graphic {
3     // Prints the graphic.
4     public void print();
5 }

1 /** "Composite" */
2 import java.util.List;
3 import java.util.ArrayList;
4 class CompositeGraphic implements Graphic {
5     // Collection of child graphics.
6     private List<Graphic> mChildGraphics = new ArrayList<Graphic>();
7     // Prints the graphic.
8     public void print() {
9         // Adds the graphic to the composition.
10        mChildGraphics.add(graphic);
11     }
12     // Removes the graphic from the composition.
13     public void remove(Graphic graphic) {
14         mChildGraphics.remove(graphic);
15     }
16     // Adds the graphic to the composition.
17     public void add(Graphic graphic) {
18         mChildGraphics.add(graphic);
19     }
20     // Prints the graphic.
21     public void print() {
22         for (Graphic graphic : mChildGraphics) {
23             graphic.print();
24         }
25     }
26     // Prints the complete graphic (four times the string "Ellipse").
27     public void print() {
28         // Adds the graphic to the composition.
29         mChildGraphics.add(graphic);
30         // Removes the graphic from the composition.
31         mChildGraphics.remove(graphic);
32     }
33 }

1 /** "Leaf" */
2 class Ellipse implements Graphic {
3     // Prints the graphic.
4     public void print() {
5         System.out.println("Ellipse");
6     }
7 }

1 /** Client */
2 public class Program {
3     public static void main(String[] args) {
4         // Initialize four ellipses
5         Ellipse ellipse1 = new Ellipse();
6         Ellipse ellipse2 = new Ellipse();
7         Ellipse ellipse3 = new Ellipse();
8         Ellipse ellipse4 = new Ellipse();
9         // Initialize three composite graphics
10        CompositeGraphic graphic = new CompositeGraphic();
11        CompositeGraphic graphic1 = new CompositeGraphic();
12        CompositeGraphic graphic2 = new CompositeGraphic();
13         // Composes the graphics
14        graphic1.add(ellipse1);
15        graphic1.add(ellipse2);
16        graphic1.add(ellipse3);
17        graphic2.add(ellipse4);
18        graphic.add(graphic1);
19        graphic.add(graphic2);
20         // Prints the complete graphic (four times the string "Ellipse").
21         graphic.print();
22     }
23 }

<?php
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()
    {
        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
{
    /**
Implementations

```php
* 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();
```

```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()
```
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
8 Decorator

The decorator pattern helps to add behavior or responsibilities to an object. This is also called “Wrapper”.

8.1 Examples

8.2 Cost

This pattern can be very expensive. You should only use it when it is really necessary. You should have lots of different behaviors and responsibilities for the same class.

8.2.1 Creation

This pattern is expensive to create.

8.2.2 Maintenance

This pattern can be expensive to maintain. If the representation of a class often changes, you will have lots of refactoring.

8.2.3 Removal

This pattern is hard to remove too.

8.3 Advises

• Put the *decorator* term in the name of the decorator classes to indicate the use of the pattern to the other developers.

8.4 Implementations

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

```csharp
using System;

// Extension methods must be parts of static classes.
static class BooleanExtensionMethodSample
{
    public static void Main()
    {
```
```csharp
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)
{
    // Return the opposite of the target.
    return !target;
}

8.5 Coffee making scenario

```
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 << "Cost: " << sample->getCost() << std::endl;
    delete sample;
}
The output of this program is given below: **Whip Milk Coffee**

Cost: 2.2
get() = 1.0

override val ingredients: String
    get() = "Coffee"
}

// Abstract decorator class - note that it implements Coffee interface
abstract class CoffeeDecorator(private val decoratedCoffee: Coffee) : Coffee {
    override// Implementing methods of the interface
    val cost: Double
        get() = decoratedCoffee.cost

    override val ingredients: String
        get() = decoratedCoffee.ingredients
}

// Decorator WithMilk mixes milk into coffee.
// Note it extends CoffeeDecorator.
class WithMilk(c: Coffee) : CoffeeDecorator(c) {
    override// Overriding methods defined in the abstract superclass
    val cost: Double
        get() = super.cost + 0.5

    override val ingredients: String
        get() = super.ingredients + ", Milk"
}

// Decorator WithSprinkles mixes sprinkles onto coffee.
// Note it extends CoffeeDecorator.
class WithSprinkles(c: Coffee) : CoffeeDecorator(c) {
    override val cost: Double
        get() = super.cost + 0.2

    override val ingredients: String
        get() = super.ingredients + ", Sprinkles"
}

The output of this program is given below:  Cost: 1.0; Ingredients: Coffee

Cost: 1.5; Ingredients: Coffee, Milk

Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles

8.6 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
Decorator

```java
// 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.

```java
// 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
    }
}
```

```java
// 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";
    }
}
```

```java
// 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:

```java
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.

### 8.7 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.

```java
// 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.

```java
// 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);
    }
}
```
Decorator

```java
// 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:

```java
public class Main {
    public static 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

8.8 Window System

```python
# the Window base class
class Window(object):
    def __init__(self, device):
        self.device = device
    def draw(self, device):
        device.append('flat window')
    def info(self):
        pass

# The decorator pattern approach
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()
```

8.8.1 Difference between subclass method and decorator pattern

```python
# The subclass approach
class BorderedWindow(Window):
    def __init__(self, device):
        super(BorderedWindow, self).__init__(device)
        device.append('borders')

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

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

def test_multi():
    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()

8.9 Coffee example

type Coffee() =
    abstract member Cost : double
    abstract member Ingredients: string list

    default this.Cost = 1.0
    default this.Ingredients = ["Coffee"]

type CoffeeDecorator(coffee: Coffee) =
    inherit Coffee ()
    override this.Cost = coffee.Cost
    override this.Ingredients = coffee.Ingredients

type WithMilk(coffee: Coffee) =
    inherit CoffeeDecorator(coffee)
    override this.Cost = base.Cost + 0.5
    override this.Ingredients = ["Milk"] |> List.append base.Ingredients

type WithSprinkles(coffee: Coffee) =
    inherit CoffeeDecorator(coffee)
    override this.Cost = base.Cost + 0.2
    override this.Ingredients = ["Sprinkles"] |> List.append base.Ingredients

let print (coffee: Coffee) =
    printfn "Cost: %.2f$; Ingredients: %A" coffee.Cost coffee.Ingredients

let withAddins = Coffee() |> WithMilk |> WithSprinkles
print withAddins

8.9.1 Dynamic languages

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

8.10 External links

- Java Design Patterns tutorial¹
- History of Design Patterns²

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

A Facade pattern hides the complexities of the system and provides an interface to the client from where the client can access the system. Dividing a system into subsystems helps reduce complexity. We need to minimize the communication and dependencies between subsystems. For this, we introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

9.1 Examples

The SCP command is a shortcut for SSH commands. A remote file copy could be done writing several commands with an SSH connection but it can be done in one command with SCP. So the SCP command is a facade for the SSH commands. Although it may not be coded in the object programming paradigm, it is a good illustration of the design pattern.

9.2 Cost

This pattern is very easy and has not additional cost.

9.2.1 Creation

This pattern is very easy to create.

9.2.2 Maintenance

This pattern is very easy to maintain.

9.2.3 Removal

This pattern is very easy to remove too.

9.3 Advises

- Do not use this pattern to mask only three or four method calls.
9.4 Implementations

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).

```csharp
/* 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();
    }
}

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();
    }
}

# 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
def start_computer
    @cpu.freeze
    @memory.load(BOOT_ADDRESS, @hard_drive.read(BOOT_SECTOR, SECTOR_SIZE))
    @cpu.jump(BOOT_ADDRESS)
    @cpu.execute
end

# Client
facade = Computer.new
facade.start_computer

# 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()

/* 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 */
```php
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();

/* 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) {
        console.log('Constant: ' + name);
    }
}

var facade = new Computer();
```

```javascript
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';
    }
};

var Computer = function () {
    var cpu, memory, hardDrive;
    cpu = new CPU();
    memory = new Memory();
    hardDrive = new HardDrive();
    var constant = function (name) {
        console.log('Constant: ' + name);
    }
}
```

125
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();

/* 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();
    }
}
}

/* Complex parts */
package intel {
    class CPU {
        def freeze() = ???
        def jump(position: Long) = ???
    }
package ram.plain {
    class Memory {
        def load(position: Long, data: Array[Byte]) = ???
    }
}

package hdd {
    class HardDrive {
        def read(lba: Long, size: Int): Array[Byte] = ???
    }
}

package pk {
    class ComputerFacade(conf: String) {
        val processor: CPU = new CPU
        val ram: Memory = new Memory
        val hd: HardDrive = new HardDrive

        val BOOT_ADDRESS: Long = ???
        val BOOT_SECTOR: Long = ???
        val SECTOR_SIZE: Int = ???

        def start() = {
            processor.freeze()
            ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE))
            processor.jump(BOOT_ADDRESS)
            processor.execute()
        }
    }
}

package common.patterns.pk.ComputerFacade

object You {
    def main(args: Array[String]) {
        new ComputerFacade("conf").start()
    }
}

program Facade;
{/$APPTYPE CONSOLE
($R *.res)
uses
    System.SysUtils;

type
    (* complex parts - > Subsystem *)
    TCPU = class

};
procedure Freeze;
  procedure Jump(position: Integer);
  procedure Execute;
end;

TMemory = class
  procedure Load(position: Integer; data: string);
end;

THardDrive = class
  function Read(lba, size: Integer): string;
end;

(* Facade *)
TComputer = class
  fCPU: TCPU;
  fMemory: TMemory;
  fHardDrive: THardDrive;
const
  BOOT_ADDRESS: Integer = 0;
  BOOT_SECTOR: Integer = 0;
  SECTOR_SIZE: Integer = 512;
public
  procedure Start_Computer;
  constructor Create;
end;

{ TCPU }
procedure TCPU.Execute;
begin
  WriteLn('CPU: execute');
end;

procedure TCPU.Freeze;
begin
  WriteLn('CPU: freese');
end;

procedure TCPU.Jump(position: Integer);
begin
  WriteLn('CPU: jump to ' + IntToStr(position));
end;

{ TMemory }
procedure TMemory.Load(position: Integer; data: string);
begin
  WriteLn('Memory: load "' + data + '" at ' + IntToStr(position));
end;

{ THardDrive }
function THardDrive.Read(lba, size: Integer): string;
begin
  WriteLn('HardDrive: read sector ' + IntToStr(lba) + ' (' + IntToStr(size) + ' bytes)');
  Result := 'hdd data';
end;

{ TComputer }
constructor TComputer.Create;
begin
  fCPU := TCPU.Create;
Facade

```pascal
fMemory := TMemory.Create;
fHardDrive := THardDrive.Create;
end;

procedure TComputer.Start_Computer;
begin
  fCPU.Freeze;
  fMemory.Load(BOOT_ADDRESS, fHardDrive.Read(BOOT_SECTOR, SECTOR_SIZE));
  fCPU.Jump(BOOT_ADDRESS);
  fCPU.Execute;
end;

var
  facad: TComputer;
begin
try
  { TODO -oUser -cConsole Main : Insert code here }
  facad := TComputer.Create;
  facad.Start_Computer;

  WriteLn(#13#10 + 'Press any key to continue...');
  ReadLn;

  facad.Free;
except
  on E: Exception do
    WriteLn(E.ClassName, ': ', E.Message);
end;
end.
```
10 Factory method

The factory pattern is a design pattern used to promote encapsulation of data representation.

Problem

We want to decide at run time what object is to be created based on some configuration or application parameter. When we write the code we do not know what class should be instantiated.

Solution

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. Its 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.

10.1 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 Java¹ code shows a very simple implementation of the factory pattern for type Foo.

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

¹ https://en.wikibooks.org/wiki/Java

131
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:

```java
// 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:

```java
// 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.

### 10.2 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):

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

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

```java
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;
    }
}
```
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:

```java
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:

```java
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:

```java
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 whatever-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:

```java
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();
}
```
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.

### 10.2.1 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:

```java
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;
        }
    }
}
```
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.

10.3 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 and narrowing casts. 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:

```java
class Foo<T> {
    // End class Foo
}
```

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

```java
class Foo<T> {
    static Foo<String> getFooOverString() {
        return new Foo<String>();
    }
    static Foo<Integer> getFooOverInteger() {
        return new Foo<Integer>();
    }
} // 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:

```java
class Foo<T> {
    public static <E> Foo<E> getFoo() {
        return new Foo<E>();
    }
} // End class Foo
```

### 10.4 Examples

In Java, a class that implements `java.sql.Connection` is a factory of statements. By calling the `createStatement()` method, you create a statement for which you only know the interface. The factory chooses the right instance class for you.

### 10.5 Cost

This pattern is not so expensive when it is implemented at the right time. It can be more expensive if you have to refactor an existing code.

#### 10.5.1 Creation

Its implementation is easy and there is no additional cost (it is not more expensive than an implementation without this pattern).

#### 10.5.2 Maintenance

There is no additional cost nor additional constraint.
10.5.3 Removal

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

10.6 Good practices

• Name the factory class with the instanciated class in prefix and the factory term in suffix, to indicate the use of the pattern to the other developers.
• Avoid to persist the factory result (in database) into the factory class, to respect the SOLID principles and allow the creation of ephemeral objects, for example into the automatic tests.

10.7 Implementation

REPORT zz_pizza_factory_test NO STANDARD PAGE HEADING .
TYPES ty_pizza_type TYPE i .
*----------------------------------------------------------------------*
* CLASS lcl_pizza DEFINITION ABSTRACT .
*----------------------------------------------------------------------*
PUBLIC SECTION .
  DATA p_pizza_name TYPE string .
  METHODS get_price ABSTRACT
      RETURNING value(y.price) TYPE i .
ENDCLASS .
*----------------------------------------------------------------------*
* CLASS lcl_ham_and_mushroom_pizza DEFINITION
*----------------------------------------------------------------------*
PUBLIC SECTION .
  METHODS constructor .
  METHODS get_price REDEFINITION .
ENDCLASS .
*----------------------------------------------------------------------*
* CLASS lcl_deluxe_pizza DEFINITION
*----------------------------------------------------------------------*
PUBLIC SECTION .
  METHODS constructor .
  METHODS get_price REDEFINITION .
ENDCLASS .
*----------------------------------------------------------------------*
* CLASS lcl_hawaiian_pizza DEFINITION
*----------------------------------------------------------------------*
PUBLIC SECTION .
  METHODS constructor .
  METHODS get_price REDEFINITION .
ENDCLASS .
*----------------------------------------------------------------------*
* 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 ,
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.

CLASS lcl_ham_and_mushroom_pizza IMPLEMENTATION.
METHOD constructor.
    super->constructor().
    p_pizza_name = 'Ham & Mushroom Pizza'(001).
ENDMETHOD.
METHOD get_price.
    y_price = 850.
ENDMETHOD.
ENDCLASS.

CLASS lcl_deluxe_pizza IMPLEMENTATION.
METHOD constructor.
    super->constructor().
    p_pizza_name = 'Deluxe Pizza'(002).
ENDMETHOD.
METHOD get_price.
    y_price = 1050.
ENDMETHOD.
ENDCLASS.

CLASS lcl_hawaiian_pizza IMPLEMENTATION.
METHOD constructor.
    super->constructor().
    p_pizza_name = 'Hawaiian Pizza'(003).
ENDMETHOD.
METHOD get_price.
    y_price = 1150.
ENDMETHOD.
ENDCLASS.

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.
    END_CASE.
ENDMETHOD.
ENDCLASS.

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:

141
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:
This C++11 implementation is based on the pre C++98 implementation in the book.

```
#include <iostream>

enum Direction {North, South, East, West};

class MapSite {
public:
    virtual void enter() = 0;
    virtual ~MapSite() = default;
};

class Room : public MapSite {
public:
    Room() :roomNumber(0) {}
    Room(int n) :roomNumber(n) {}
    void setSide(Direction d, MapSite* ms) {
        std::cout << "Room::setSide " << d << ' ' << ms << '
';
    }
    virtual void enter() {}
    Room(const Room&) = delete; // rule of three
    Room& operator=(const Room&) = delete;
private:
    int roomNumber;
};

class Wall : public MapSite {
public:
    Wall() {}
    virtual void enter() {}
};

class Door : public MapSite {
public:
    Door(Room* r1 = nullptr, Room* r2 = nullptr)
        :room1(r1), room2(r2) {}
    virtual void enter() {}
    Door(const Door&) = delete; // rule of three
    Door& operator=(const Door&) = delete;
private:
    Room* room1;
    Room* room2;
};

class Maze {
public:
    void addRoom(Room* r) {
        std::cout << "Maze::addRoom " << r << '
';
    }
    Room* roomNo(int) const {
        return nullptr;
    }
};

// If createMaze calls virtual functions instead of constructor calls to create
// the rooms, walls, and doors it requires, then you can change the classes that
// get instantiated by making a subclass of MazeGame and redefining those virtual
// functions. This approach is an example of the Factory Method (121) pattern.

class MazeGame {
public:

```
Factory method

```cpp
Maze* createMaze () {
    Maze* aMaze = makeMaze();
    Room* r1 = makeRoom(1);
    Room* r2 = makeRoom(2);
    Door* theDoor = makeDoor(r1, r2);
    aMaze->addRoom(r1);
    aMaze->addRoom(r2);
    r1->setSide(North, makeWall());
    r1->setSide(East, theDoor);
    r1->setSide(South, makeWall());
    r1->setSide(West, makeWall());
    r2->setSide(North, makeWall());
    r2->setSide(East, makeWall());
    r2->setSide(South, makeWall());
    r2->setSide(West, theDoor);
    return aMaze;
}

// factory methods:
virtual Maze* makeMaze() const {
    return new Maze;
}
virtual Room* makeRoom(int n) const {
    return new Room(n);
}
virtual Wall* makeWall() const {
    return new Wall;
}
virtual Door* makeDoor(Room* r1, Room* r2) const {
    return new Door(r1, r2);
}
virtual ~MazeGame() = default;
}

int main() {
    MazeGame game;
    game.createMaze();
}
```

The program output is like:

```
Maze::addRoom 0xcaced0
Maze::addRoom 0xcacef0
Room::setSide 0 0xcad340
Room::setSide 2 0xcacf10
Room::setSide 1 0xcad360
Room::setSide 3 0xcad380
Room::setSide 0 0xcad3a0
Room::setSide 2 0xcad3c0
Room::setSide 1 0xcad3e0
Room::setSide 3 0xcacf10
```

In Common Lisp\(^2\), factory methods are not really needed, because classes and class names are first class values.

```lisp
(defun pizza ()
  (price :accessor price))
(defun ham-and-mushroom-pizza (pizza)
  ((price :uniform 850)))
(defun deluxe-pizza (pizza)

\(^2\) https://en.wikibooks.org/wiki/Common%20Lisp

144
((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

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();
10.8 Example with pizza

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

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

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.");
}

Output:

Price of HamMushroom is 850
Price of Deluxe is 1050
Price of Hawaiian is 1150

10.9 Another example with image

Abstract creator

Interface to create the Product.

```java
package mypkg;
import java.awt.image.BufferedImage;
import java.io.IOException;
/**
 * @author xxx
 */
public interface PhotoReader {
    public BufferedImage getImage() throws IOException;
}
```

Concrete creator

a class to create specific Product.

```java
package mypkg;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
```
import javax.imageio.ImageIO;
import javax.imageio.ImageReader;
import javax.imageio.stream.ImageInputStream;

/**
 * @author xxx
 */
public class JPEGReader implements PhotoReader {
    ImageReader reader;
    File jpegFile;
    ImageInputStream iis;
    public JPEGReader(String filePath) throws IOException {
        jpegFile = new File(filePath);
        iis = ImageIO.createImageInputStream(jpegFile);
        Iterator readers = ImageIO.getImageReadersByFormatName("jpg");
        reader = (ImageReader)readers.next();
        this.reader.setInput(iis, true);
    }
    public BufferedImage getImage() throws IOException {
        return reader.read(0);
    }
}

Factory class

a class to return a specific concrete creator at runtime to create the product.
This example in JavaScript\(^3\) uses Firebug\(^4\) console to output information.

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

3  https://en.wikibooks.org/wiki/JavaScript
4  https://en.wikibooks.org/wiki/Firebug%20%28web%20development%29

149
Factory method

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

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)

Factorys 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 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;
}
Output

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

```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\n", $p, PizzaFactory::createPizza($p)->getPrice());
}
// Output:
// Price of HamAndMushroom is 8.50
// Price of Deluxe is 10.50
// Price of Hawaiian is 11.50
?>

#
# 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:

    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:
```python
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.

```csharp
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
```

153
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
11 Flyweight

It is a mechanism by which you can avoid creating a large number of object instances to represent the entire system. To decide if some part of a program is a candidate for using Flyweights, consider whether it is possible to move some data from the class and make it extrinsic.

11.1 Examples

A classic example usage of the flyweight pattern is the data structures for graphical representation of characters in a word processor. It might be desirable to have, for each character in a document, a glyph object containing its font outline, font metrics, and other formatting data, but this would amount to hundreds or thousands of bytes for each character. Instead, for every character there might be a reference to a flyweight glyph object shared by every instance of the same character in the document; only the position of each character (in the document and/or the page) would need to be stored internally. Another example is string interning\(^1\).

---

\(^1\) https://en.wikipedia.org/wiki/String%20interning
In video games, it is usual that you have to display the same sprite (i.e. an image of an item of the game) several times. It would highly use the CPU and the memory if each sprite was a different object. So the sprite is created once and then is rendered at different locations in the screen. This problem can be solved using the flyweight pattern. The object that renders the sprite is a flyweight.

### 11.2 Cost

There are several implementations for this pattern. So it's up to you to find a cheap implementation. Only implement this pattern if you have or will have CPU or memory issues.

#### 11.2.1 Creation

This pattern is quite easy to create.

#### 11.2.2 Maintenance

This pattern is quite easy to maintain.
11.2.3 Removal

This pattern is quite easy to remove too.

11.3 Advises

- Use pre-existing tools from the language like the sets in Java.

11.4 Implementations

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.

```java
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 WeakHashMap<FontData, WeakReference<FontData>>
        FLY_WEIGHT_DATA =
        new WeakHashMap<FontData, WeakReference<FontData>>();
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;
    private FontData(int pointSize, String fontFace, Color color,
        EnumSet<FontEffect> effects) {
        this.pointSize = pointSize;
        this.fontFace = fontFace;
        this.color = color;
        this.effects = Collections.unmodifiableSet(effects);
    }
    public static FontData create(int pointSize, String fontFace, Color color,
        FontEffect... effects) {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.class);
        for (FontEffect fontEffect : effects) {
            effectsSet.add(fontEffect);
        }
        FontData data = new FontData(pointSize, fontFace, color, effectsSet);
        FontData result = null;
        // Retrieve previously created instance with the given values if it
        // (still) exists
        WeakReference<FontData> 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 WeakReference<FontData> (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.

'''
http://codesnipers.com/?q=python-flyweights'''

from __future__ import print_function
import weakref
class Card(object):
    # comment __new__ and uncomment __init__ to see the difference
    '''The object pool. Has built-in reference counting'''
    _CardPool = weakref.WeakValueDictionary()

    '''If the object exists in the pool just return it (instead of creating a
    new one)'''
    def __new__(cls, value, suit):
        obj = Card._CardPool.get(value + suit, None)
        if not obj:
            obj = object.__new__(cls)
            Card._CardPool[value + suit] = obj
        obj.value, obj.suit = value, suit
        return obj

    # def __init__(self, value, suit):
    #     self.value, self.suit = value, suit

    def __repr__(self):
        return '<Card: %s%s>' % (self.value, self.suit)

if __name__ == '__main__':
c1 = Card('9', 'h')
c2 = Card('9', 'h')
print(c1, c2)
print(c1 == c2)
print(id(c1), id(c2))
12 Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

12.1 Examples

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.

12.2 Cost

This pattern is not expensive. It dramatically reduces the business code so there is only few code to handle.

\(^1\) https://en.wikipedia.org/wiki/Reverse%20Polish%20notation
12.2.1 Creation
If the code already exists, this pattern is a little expensive.

12.2.2 Maintenance
This pattern is very easy to maintain. There is no additional cost due to the pattern.

12.2.3 Removal
This pattern is easy to remove with refactoring operations from your IDE.

12.3 Advises
• Use the Interpreter term to indicate the use of the pattern to the other developers.

12.4 Implementations
The following Backus–Naur form 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'
nnumber ::= digit | digit number
```

defines a language that contains Reverse Polish Notation\(^2\) expressions like:

```
a b +
a b c * -
a b + c a - -
```

This structural code demonstrates the Interpreter patterns, which using a defined grammar, provides the interpreter that processes parsed statements.

```csharp
using System;
using System.Collections.Generic;
namespace OOP;

class Program
{
    static void Main()
    {
        var context = new Context();
        var input = new MyExpression();
        var expression = new OrExpression
        {
```

\(^2\) https://en.wikibooks.org/wiki/Reverse%20Polish%20Notation
Left = new EqualsExpression
{
    Left = input,
    Right = new MyExpression { Value = "4" }
},
Right = new EqualsExpression
{
    Left = input,
    Right = new MyExpression { Value = "four" }
};

input.Value = "four";
expression.Interpret(context);
// Output: "true"
Console.WriteLine(context.Result.Pop());

input.Value = "44";
expression.Interpret(context);
// Output: "false"
Console.WriteLine(context.Result.Pop());
}
}

class Context
{
    public Stack<string> Result = new Stack<string>();
}

interface IExpression
{
    void Interpret(Context context);
}

abstract class OperatorExpression : IExpression
{
    public IExpression Left { private get; set; }
    public IExpression Right { private get; set; }

    public void Interpret(Context context)
    {
        Left.Interpret(context);
        string leftValue = context.Result.Pop();

        Right.Interpret(context);
        string rightValue = context.Result.Pop();

        DoInterpret(context, leftValue, rightValue);
    }

    protected abstract void DoInterpret(Context context, string leftValue, string rightValue);
}

class EqualsExpression : OperatorExpression
{
    protected override void DoInterpret(Context context, string leftValue, string rightValue)
    {
        context.Result.Push(leftValue == rightValue ? "true" : "false");
    }
}

class OrExpression : OperatorExpression
{
    protected override void DoInterpret(Context context, string leftValue,
class MyExpression : IExpression
{
    public string Value { private get; set; }

    public void Interpret(Context context)
    {
        context.Result.Push(Value);
    }
}

Another example:

using System;
using System.Collections.Generic;

namespace Interpreter
{
    class Program
    {
        interface IExpression
        {
            int Interpret(Dictionary<string, int> variables);
        }

        class Number : IExpression
        {
            public int number;
            public Number(int number) { this.number = number; }
            public int Interpret(Dictionary<string, int> variables) { return number; }
        }

        abstract class BasicOperation : IExpression
        {
            IExpression leftOperator, rightOperator;

            public BasicOperation(IExpression left, IExpression right)
            {
                leftOperator = left;
                rightOperator = right;
            }

            public int Interpret(Dictionary<string, 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(Dictionary<string, int> variables)
    {
        return variables[name];
    }
}

class Evaluator
{
    private IExpression syntaxTree;
    public Evaluator(string expression)
    {
        Stack<IExpression> stack = new Stack<IExpression>();
        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(Dictionary<string, int> context)
    {
        return syntaxTree.Interpret(context);
    }
}

static void Main(string[] args)
{
    Evaluator evaluator = new Evaluator("w x z - +");
    Dictionary<string, int> values = new Dictionary<string, int>();
    values.Add("w", 5);
    values.Add("x", 10);
    values.Add("z", 42);
    Console.WriteLine(evaluator.Evaluate(values));
}
public class Interpreter {
    @FunctionalInterface
    public interface Expr {
        int interpret(Map<String, Integer> context);
    }

    static Expr number(int number) {
        return context -> number;
    }

    static Expr plus(Expr left, Expr right) {
        return context -> left.interpret(context) +
               right.interpret(context);
    }

    static Expr minus(Expr left, Expr right) {
        return context -> left.interpret(context) -
               right.interpret(context);
    }

    static Expr variable(String name) {
        return context -> context.getOrDefault(name, 0);
    }
}

While the interpreter pattern does not address parsing, a parser is provided for completeness.

private static Expr parseToken(String token, ArrayDeque<Expr> stack) {
    Expr left, right;
    switch(token) {
        case "+":
            // It's necessary to remove first the right operant from the stack
            right = stack.pop();
            // ...and then the left one
            left = stack.pop();
            return Expr.plus(left, right);
        case "-":
            right = stack.pop();
            left = stack.pop();
            return Expr.minus(left, right);
        default:
            return Expr.variable(token);
    }
}

public static Expr parse(String expression) {
    ArrayDeque<Expr> stack = new ArrayDeque<Expr>();
    for (String token : expression.split(" ")) {
        stack.push(parseToken(token, stack));
    }
    return stack.pop();
}

Finally evaluating the expression "w x z - +" with w = 5, x = 10, and z = 42.

public static void main(final String[] args) {
    Expr expr = parse("w x z - +");
    Map<String, Integer> context = Map.of("w", 5, "x", 10, "z", 42);
    int result = expr.interpret(context);
    System.out.println(result);  // -27
}

Another example:
Implementations

```java
import java.util.Map;
interface Expression {
    public int interpret(Map<String, Expression> variables);
}

class Number implements Expression {
    private int number;
    public Number(int number) {
        this.number = number;
    }
    public int interpret(Map<String, Expression> variables) {
        return number;
    }
}

class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }
    public int interpret(Map<String, Expression> variables) {
        return leftOperand.interpret(variables) +
               rightOperand.interpret(variables);
    }
}

class Minus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Minus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }
    public int interpret(Map<String, Expression> variables) {
        return leftOperand.interpret(variables) -
               rightOperand.interpret(variables);
    }
}

class Variable implements Expression {
    private String name;
    public Variable(String name) {
        this.name = name;
    }
    public int interpret(Map<String, 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) {
        Stack<Expression> expressionStack = new Stack<Expression>();
        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(Map<String,Expression> context) {
        return syntaxTree.interpret(context);
    }
}

Finally evaluating the expression "wxz-+" 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);
        Map<String,Expression> variables = new HashMap<String,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);
    }
}

As JavaScript is dynamically typed, we do not implement an interface.

// Nonterminal expression
class Plus {
    a;
    b;
    constructor(a, b) {
        this.a = a;
        this.b = b;
    }
    interpret(context) {
        return this.a.interpret(context) + this.b.interpret(context);
    }
}

// Nonterminal expression
class Minus {
  a;
  b;
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }
  interpret(context) {
    return this.a.interpret(context) - this.b.interpret(context);
  }
}

// Nonterminal expression
class Times {
  a;
  b;
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }
  interpret(context) {
    return this.a.interpret(context) * this.b.interpret(context);
  }
}

// Nonterminal expression
class Divide {
  a;
  b;
  constructor(a, b) {
    this.a = a;
    this.b = b;
  }
  interpret(context) {
    return this.a.interpret(context) / this.b.interpret(context);
  }
}

// Terminal expression
class Number {
  a;
  constructor(a, b) {
    this.a = a;
  }
  interpret(context) {
    return this.a;
  }
}

// Terminal expression
class Variable {
  a;
  constructor(a) {
    this.a = a;
  }
  interpret(context) {
    return context[this.a] || 0;
  }
}

// Client
class Parse {
  context;
  constructor(context) {
    this.context = context;
  }
  parse(expression) {
    let tokens = expression.split(" ");
    let queue = [];
    for (let token of tokens) {
      switch (token) {
      }
```javascript
var res = new Parse({v: 45}).parse("16 v * 76 22 - -");
console.log(res);
//666
```

**Example 1:**

```typescript
/**
 * AbstractExpression
 */
interface Expression {
    public function interpret(array $context): int;
}

/**
 * TerminalExpression
 */
class TerminalExpression implements Expression {
    /** @var string */
    private $name;
    public function __construct(string $name) {
        
    }
```
```php
$this->name = $name;

public function interpret(array $context): int
{
    return intval($context[$this->name]);
}
```
/**
 * Client
 */
class InterpreterClient {
    protected function parseList(array &$stack, array $list, int &$index)
    {
        /** @var string $token */
        $token = $list[$index];
        switch($token) {
            case '-':
                list($left, $right) = $this->fetchArguments($stack, $list, $index);
                return new MinusExpression($left, $right);
            case '+':
                list($left, $right) = $this->fetchArguments($stack, $list, $index);
                return new PlusExpression($left, $right);
            default:
                return new TerminalExpression($token);
        }
    }

    protected function fetchArguments(array &$stack, array $list, int &$index): array
    {
        /** @var Expression $left */
        $left = array_pop($stack);
        /** @var Expression $right */
        $right = array_pop($stack);
        if ($right === null) {
            ++$index;
            $this->parseListAndPush($stack, $list, $index);
            $right = array_pop($stack);
        }
        return array($left, $right);
    }

    protected function parseListAndPush(array &$stack, array $list, int &$index)
    {
        array_push($stack, $this->parseList($stack, $list, $index));
    }

    protected function parse(string $data): Expression
    {
        $stack = [];
        $list = explode(' ', $data);
        for ($index=0; $index<count($list); $index++) {
            $this->parseListAndPush($stack, $list, $index);
        }
        return array_pop($stack);
    }

    public function main()
    {
        $data = "u + v - w + z";
        $expr = $this->parse($data);
        $context = ['u' => 3, 'v' => 7, 'w' => 35, 'z' => 9];
        $res = $expr->interpret($context);
        echo "result: $res" . PHP_EOL;
    }
}
// test.php

function loadClass($className)
{
    require_once __DIR__ . "/$className.php";
}

spl_autoload_register('loadClass');

(new InterpreterClient())->main();
//result: -16

Example 2: Based on the example above with another realization of the client.

```php
/**
 * Client
 */

class InterpreterClient
{

    public function parseToken(string $token, array &$stack): Expression
    {
        switch($token) {
            case '-':
                /** @var Expression $left */
                $left = array_pop($stack);
                /** @var Expression $right */
                $right = array_pop($stack);
                return new MinusExpression($left, $right);
            case '+':
                /** @var Expression $left */
                $left = array_pop($stack);
                /** @var Expression $right */
                $right = array_pop($stack);
                return new PlusExpression($left, $right);
            default:
                return new TerminalExpression($token);
        }
    }

    public function parse(string $data): Expression
    {
        $unfinishedData = null;
        $stack = [];
        $list = explode(' ', $data);
        foreach ($list as $token) {
            $data = $this->parseToken($token, $stack);
            if (($unfinishedData instanceof NonTerminalExpression) &&
                ($data instanceof TerminalExpression)) {
                $unfinishedData->setRight($data);
                array_push($stack, $unfinishedData);
                $unfinishedData = null;
                continue;
            } elseif ($data instanceof NonTerminalExpression) {
                if ($data->getRight() === null) {
                    $unfinishedData = $data;
                    continue;
                }
            }
            array_push($stack, $data);
        }
        return $unfinishedData;
    }
}
```

171
Example 3: 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
<?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");
        }
    }

    $data = "u + v - w + z";
    $expr = $this->parse($data);
    $context = [‘u’ => 3, ‘v’ => 7, ‘w’ => 35, ‘z’ => 9];
    $res = $expr->interpret($context);
    echo "result: $res" . PHP_EOL;
```
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:
$expression = "6 10 42 - +";
// or for it:
//$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).
13 Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

13.1 Examples

In Java, the interface `java.util.Iterator<E>` is an implementation of the iterator pattern. That way, all the objects that implement the `java.lang.Iterable<T>` interface don't need a handy implementation of this pattern.

13.2 Cost

This pattern has a cost. Only implement this pattern for an important amount of code. IDE refactoring can't help you much.

13.2.1 Creation

This pattern has a cost to create.

13.2.2 Maintenance

This pattern is easy to maintain.

13.2.3 Removal

This pattern has a cost to remove too.

13.3 Advises

- Put the `iterator` term in the name of the iterator class to indicate the use of the pattern to the other developers.

13.4 Implementations

Java has the `Iterator` interface.

A simple example showing how to return integers between `[start, end]` using an `Iterator`
import java.util.Iterator;
import java.util.NoSuchElementException;

public class RangeIteratorExample {
    public static Iterator<Integer> range(int start, int end) {
        return new Iterator<>() {
            private int index = start;
            @Override
            public boolean hasNext() {
                return index < end;
            }
            @Override
            public Integer next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return index++;
            }
        };
    }
    public static void main(String[] args) {
        var iterator = range(0, 10);
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
        // or using a lambda
        iterator.forEachRemaining(System.out::println);
    }
}

As of Java 5, objects implementing the java/langIterable interface, which returns an Iterator from its only method, can be traversed using Java's foreach loop\(^1\) syntax. The java/utilCollection interface from the Java collections framework\(^2\) extends Iterable. Example of class Family implementing the Iterable interface:

```
import java.util.Iterator;
import java.util.Set;

class Family<E> implements Iterable<E> {
    private final Set<E> elements;
    public Family(Set<E> elements) {
        this.elements = Set.copyOf(elements);
    }
    @Override
    public Iterator<E> iterator() {
        return elements.iterator();
    }
}
```

The class `IterableExample` demonstrates the use of class `Family`:

```java
public class IterableExample {
    public static void main(String[] args) {
        var weasleys = Set.of(
            "Percy", "Fred", "George", "Ron", "Ginny"
        );
        var family = new Family<>(weasleys);
        for (var name : family) {
            System.out.println(name + " Weasley");
        }
    }
}
```

Output:

```
Ron Weasley
Molly Weasley
Percy Weasley
Fred Weasley
Charlie Weasley
George Weasley
Arthur Weasley
Ginny Weasley
Bill Weasley
```

Here is another example in Java³:

```java
import java.util.BitSet;
import java.util.Iterator;
public class BitSetIterator implements Iterator<Boolean> {
    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:

```java
import java.util.BitSet;
public class TestClientBitSet {
    public static void main(String[] args) {
```

// create BitSet and set some bits
BitSet bitset = new BitSet();
set.bitset(1);
set.bitset(3400);
set.bitset(20);
set.bitset(47);
for (BitSetIterator iter = new BitSetIterator(bitset); iter.hasNext(); )
{
    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) {
        Collection<Object> al = new ArrayList<Object>();
al.add(new Integer(42));
al.add("test");
al.add(new Double("-12.34"));
for (Iterator<Object> iter = al.iterator(); iter.hasNext(); ) {
    System.out.println(iter.next());
}
for (Object o : al) {
    System.out.println(o);
    }
    }
    }

JavaScript\(^4\), as part of ECMAScript 6, supports the iterator pattern with any object that provides a \texttt{next()} method, which returns an object with two specific properties: \texttt{done} and \texttt{value}. Here’s an example that shows a reverse array iterator:

```javascript
function reverseArrayIterator(array) {
    var index = array.length - 1;
    return {
        next: () =>
        index >= 0 ?
        {value: array[index--], done: false} :
        {done: true}
    }
}
const it = reverseArrayIterator(['three', 'two', 'one']);
console.log(it.next().value); //-> 'one'
console.log(it.next().value); //-> 'two'
console.log(it.next().value); //-> 'three'
console.log('Are you done? ${it.next().done}'); //-> true
```

Most of the time, though, it is desirable to provide \texttt{Iterator}\(^5\) semantics on objects so that they can be iterated automatically via \texttt{for...of} loops. Some of JavaScript’s built-in types such as \texttt{Array}, \texttt{Map}, or \texttt{Set} already define their own iteration behavior. The same effect

\[^4\] https://en.wikibooks.org/wiki/JavaScript
\[^5\] Iterators and generators "\*= Retrieved"
Implementations can be achieved by defining an object's meta ` @@iterator ` method, also referred to by ` Symbol.iterator `. This creates an Iterable object.

Here's an example of a range function that generates a list of values starting from ` start ` to ` end `, exclusive, using a regular ` for ` loop to generate the numbers:

```javascript
function range(start, end) {
    return {
        [Symbol.iterator]() { // #A
            return this;
        },
        next() {
            if (start < end) {
                return { value: start++, done: false }; // #B
            }
            return { done: true, value: end }; // #B
        }
    }
}

for (number of range(1, 5)) {
    console.log(number); // -> 1, 2, 3, 4
}
```

The iteration mechanism of built-in types, like strings, can also be manipulated:

```javascript
let iter = ['I', 't', 'e', 'r', 'a', 't', 'o', 'r'][Symbol.iterator]();
iter.next().value; //-> I
iter.next().value; //-> t
```


C#[8] statement ` foreach ` is designed to easily iterate through the collection that implements ` System.Collections.IEnumerator ` and/or ` System.Collections.Generic.IEnumerator<T> ` interface. Since C# v2, ` foreach ` is also able to iterate through types that implement ` System.Collections.Generic.IEnumerable<T> ` and ` System.Collections.Generic.IEnumerator<T> `.[9]

Example of using ` foreach ` statement:

```csharp
var primes = new List<int>{ 2, 3, 5, 7, 11, 13, 17, 19 }; long m = 1;
foreach (var prime in primes)
    m *= prime;
```

Here is another example in C#[10]:

---

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();
    }
}

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); }
    }
}

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)
    {
PHP\(^{11}\) supports the iterator pattern via the Iterator interface, as part of the standard distribution.\(^{12}\) Objects that implement the interface can be iterated over with the `foreach` language construct.

Example of patterns using PHP:

```php
<?php

// BookIterator.php
namespace DesignPatterns;

class BookIterator implements \

```
Iterator

```php
public function rewind(): void
{
    $this->i_position = 0;
}

public function valid(): bool
{
    return !is_null($this->booksCollection->getTitle($this->i_position));
}
```

```php
<?php
// BookCollection.php
namespace DesignPatterns;

class BookCollection implements \IteratorAggregate
{
    private array $a_titles = array();

    public function getIterator()
    {
        return new BookIterator($this);
    }

    public function addTitle(string $string): void
    {
        $this->a_titles[] = $string;
    }

    public function getTitle($key)
    {
        if (isset($this->a_titles[$key]))
        {
            return $this->a_titles[$key];
        }
        return null;
    }

    public function is_empty(): bool
    {
        return empty($this->$a_titles);
    }
}
```

```php
// index.php
require 'vendor/autoload.php';
use DesignPatterns\BookCollection;

$booksCollection = new BookCollection();
$booksCollection->addTitle('Design Patterns');
$booksCollection->addTitle('PHP7 is the best');
$booksCollection->addTitle('Laravel Rules');
$booksCollection->addTitle('DHH Rules');

foreach ($booksCollection as $book) {
    var_dump($book);
}
x
```
Another example: As a default behavior in PHP\textsuperscript{14}, 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:

```php
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. In Perl\textsuperscript{15}, objects providing an iterator interface either overload\textsuperscript{16} the `<>` (iterator operator),\textsuperscript{17} or provide a hash or tied hash\textsuperscript{18} interface that can be iterated over with `each`\textsuperscript{19, 20}.

Both `<>` and `each` return `undef` when iteration is complete. Overloaded `<>` operator:

```perl
# fibonacci sequence
package FibIter;
use overload
  '<>' => 'next_fib';
sub new {
  my $class = shift;
  # ...
}
```

---

15. [https://en.wikibooks.org/wiki/Perl](https://en.wikibooks.org/wiki/Perl)
17. File handle objects implement this to provide line by line reading of their contents
18. [http://perldoc.perl.org/Tie/Hash.html](http://perldoc.perl.org/Tie/Hash.html)
19. [http://perldoc.perl.org/Tie/Array.html](http://perldoc.perl.org/Tie/Array.html)
20. In Perl 5.12, arrays and tied arrays [http://perldoc.perl.org/Tie/Array.html](http://perldoc.perl.org/Tie/Array.html) can be iterated over like hashes, with `each`
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 (notfileno $self->{fh}) {
        open $fh, '<', $self->{fname} or die "open $!";
    }
    # reset file pointer
    seek( $fh, 0, 0 );
    chomp(my $line = <$fh>);
    $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) = @_;
Python prescribes a syntax for iterators as part of the language itself, so that language keywords such as `for` work with what Python calls iterables. An iterable has an `__iter__()` method that returns an iterator object. The "iterator protocol" requires `next()` return the next element or raise a `StopIteration` exception upon reaching the end of the sequence. Iterators also provide an `__iter__()` method returning themselves so that they can also be iterated over; e.g., using a `for` loop. Generators are available since 2.2.

In Python 3, `next()` was renamed `__next__()`.

In Python 3, 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:

```plaintext
# 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
```

Raku provides APIs for iterators, as part of the language itself, for objects that can be iterated with `for` and related iteration constructs, like assignment to a Positional

---

22 https://en.wikibooks.org/wiki/Python%20Programming

185
An iterable must at least implement an iterator method that returns an iterator object. The "iterator protocol" requires the pull-one method to return the next element if possible, or return the sentinel value IterationEnd if no more values could be produced. The iteration APIs is provided by composing the Iterable role, Iterator, or both, and implementing the required methods.

To check if a type object or an object instance is iterable, the does method can be used:

```
say Array.does(Iterable); #=> True
say [1, 2, 3].does(Iterable); #=> True
say Str.does(Iterable); #=> False
say "Hello".does(Iterable); #=> False
```

The does method returns True if and only if the invocant conforms to the argument type.

Here's an example of a range subroutine that mimics Python's range function:

```raku
def multi range(Int:D $start, Int:D $end where * <= $start, Int:D $step where * < 0 = -1) { (class {
  also does Iterable does Iterator;
  has Int ($.start, $.end, $.step);
  has Int $!count = $!start;

  method iterator ( self )
  method pull-one(--> Mu) {
    if $!count > $!end {
      my $i = $!count;
      $!count += $!step;
      return $i;
    }
    else {
      $!count = $!start;
      return IterationEnd;
    }
  }
 }).new(:$start, :$end, :$step)
}
```

```raku
def multi range(Int:D $start, Int:D $end where * >= $start, Int:D $step where * > 0 = 1) { (class {
  also does Iterable does Iterator;
  has Int ($.start, $.end, $.step);
  has Int $!count = $!start;

  method iterator ( self )
  method pull-one(--> Mu) {
    if $!count < $!end {
      my $i = $!count;
      $!count += $!step;
      return $i;
    }
    else {
      $!count = $!start;
      return IterationEnd;
    }
  }
 }).new(:$start, :$end, :$step)
```
MATLAB 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,

```plaintext
% Define an array of integers
myArray = [1, 3, 5, 7, 11, 13];
for n = myArray
    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,

```plaintext
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
end
```

```plaintext
function outScalar = myCustomFun(inScalar)
    % Simply multiply by 2
    outScalar = 2*inScalar;
end
```

---

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)\(^{29}\). This is written in the new class-definition syntax introduced with MATLAB software version 7.6 (R2008a)\(^{30}\) 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.

### 13.5 References


\(^{30}\) New Class-Definition Syntax Introduced with MATLAB Software Version 7.6\(^ {31}\). The MathWorks, Inc. Retrieved September 22, 2009
14 Mediator

This pattern defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. 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.

14.1 Participants

**Mediator** — defines the interface for communication between *Colleague* objects.

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

**ConcreteColleague** — communicates with other *Colleagues* through its *Mediator*.

![Diagram](image)

Figure 19

14.2 Examples

14.3 Cost

This pattern is not expensive because there is not much constraint on it.

14.3.1 Creation

This pattern is easy to create.
14.3.2 Maintenance

This pattern is easy to maintain.

14.3.3 Removal

Each object can easily be transformed to another structure using refactoring.

14.4 Advises

- Put the mediator term in the name of the iterator class to indicate the use of the pattern to the other developers.

14.5 Implementations

```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(ActionEvent 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));
    }
}

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 cmd = (Command) ae.getSource();
        cmd.execute();
    }
    public static void main(String[] args) {
        new MediatorDemo();
    }
}

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)
    {
Mediator

```csharp
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);
    }
}
```
15 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.

The following Java¹ program illustrates the "undo" usage of the Memento Pattern.

```java
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);
    }
}

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) {
        List<Originator.Memento> savedStates = new ArrayList<Originator.Memento>();
        Originator originator = new Originator();
```

¹ https://en.wikibooks.org/wiki/Subject%3AJava%20programming%20language
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

15.1 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;
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()
{

Another way to implement memento in C#:

```csharp
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(IEnumerable<IShape> shapes)
    {
        foreach(IShape shape in shapes)
        {
            shape.Draw();
        }
    }

    public void MoveAndScale(IEnumerable<IShape> shapes)
    {
        foreach(IShape shape in shapes)
        {
            shape.Scale(10);
            shape.Move(3, 2);
        }
    }

    public IEnumerable<IMemento> SaveStates(IEnumerable<IShape> shapes)
    {
        List<IMemento> states = new List<IMemento>();
        foreach(IShape shape in shapes)
        {
            states.Add(state.GetState());
        }
        return states;
    }

    public void RestoreStates(IEnumerable<IMemento> 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
        IEnumerable<IMemento> 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)
Another way to implement memento in C#

```csharp
{ 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); end;

TCaretaker = class
private
  _memento: TMemento;
public
  property Memento: TMemento read _memento write _memento;
end;

constructor TMemento.Create(stateToSave: string);
begin
  _state := stateToSave;
end;

function TMemento.GetSavedState: string;
begin
  writeln('restoring state from Memento');
  result := _state;
end;

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;
```

201
Memento

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.

import java.util.Date
import common.patterns.Originator.Memento
import scala.collection.immutable.TreeMap

object Originator {
  case class Memento(x: Int, y: Int, time: Date)
}

case class Originator(var x: Int, var y: Int) {
  def setState(x: Int, y: Int) {
    this.x = x
    this.y = y
  }

  def createMemento(): Memento = Memento(x, y, new Date())

  def restoreFromMemento(restore: Memento) {
    this.x = restore.x
    this.y = restore.y
  }
}

object Caretaker extends App {
  var states: TreeMap[Date, Memento] = new TreeMap
  val originator = new Originator(0, 0)
  var memento = originator.createMemento()
  states += (memento.time -> memento)

  Thread.sleep(100)
  originator.setState(5, 4)
  memento = originator.createMemento()
Another way to implement memento in C#:

```csharp
states += (memento.time -> memento)
Thread.sleep(100)
originator.setState(10, 10)
memento = originator.createMemento()
states += (memento.time -> memento)

originator.restoreFromMemento(states(states.drop(1).firstKey)) //restore
second state
println(originator)
```

Output:

```
Originator(5,4)
```
16 Model–view–controller

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. 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 sees 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.

16.1 Model

Generally, the model is constructed first. The model has two jobs: it must both store a state and manage subscribers. The state does not need to be anything special; you simply need to define how you’re going to store data, with setters and getters. However, anything which can change (any property) must have a list of listeners which it contacts whenever the value changes. The property listeners allow us to have multiple views on the same data, which are all live-updated when the state changes. The code for this can usually be defined in a superclass and inherited, so that you just write it in one place and then it applies consistently to every property. Nested states (for example, dynamically configured properties) may require some extra thinking about how you want to report the changes.

\(^{1}\) [https://en.wikipedia.org/wiki/Graphical%20user%20interface](https://en.wikipedia.org/wiki/Graphical%20user%20interface)
People sometimes use the word 'model' to refer to the 'data model' – the architecture of the data structures in the application. A good MVC framework takes care of the model superclass and subscriptions for you, so that you can simply define your own data structure in that framework's model-language, and then immediately build the other components. In this end-user sense, defining the data model is equivalent to defining a model. However, if you are designing a framework, it is not MVC if it lacks property listeners.

### 16.2 View

Once you write a data model, the next easiest thing to write is usually a view. The view is the part of the application which subscribes to a model. Usually it presents it to a *user* alongside a *user interface*, or GUI. The GUI contains other components too, which are usually part of the *controller* and can be handled later.

You can also have view-controller components which have nothing to do with user interfaces. You can imagine, for example, an MVC circuit-board application where a model simply manages numbers (voltages and currents) throughout the circuit. A resistor, for example, then has a "view" of the voltages at each end, tightly coupled to a "controller" which updates the current going through that resistor. All of the components talking to each other through the model would then eventually perform the circuit's actions in real-time.

When you're writing a view, there are two things to think about: "what do I do when the state changes?" and "how do I display this to the user?" Your MVC framework usually provides *editors* for various properties in the model, like date selectors, text fields, sliding bars, and combo boxes\(^\text{2}\). More complex properties often require more complex views and editors, like tree views. Your MVC framework will probably have already figured out how to connect these components to the model, and might even auto-generate appropriate components based on the type of a given property in the model.

### 16.3 Controller

The rest of the GUI – the parts which do not update when the model changes – are the responsibility of the controller. This includes navigating around the view, as well as what you do when someone tries to edit the data in the view. Strictly speaking, a view cannot be edited and is 'read-only' – when you try to modify a field in the view, the controller needs to pick up the editing event, process it, and send it to the model; the model will then update the view if/when the value actually changes.

Different frameworks handle controllers in different ways. Some frameworks allow callback functions to be associated with editors, which get called when the value is changed. Other frameworks 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 loses focus, etc. In many frameworks, the controller appears as a collection of methods and listeners built into both the data model and the view. For example, if you have a "password" property

in the model which needs to be salted\(^3\) and hashed\(^4\) before being stored, this logic might appear in a `setPassword` function living in the data model. Similarly, when a framework generates a view, the widgets for the view are often not read-only but rather read-write, with listeners. The actual controllers provided by the MVC framework then "plug into" each of these interfaces, and pass data between them.

### 16.3.1 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 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.

In Java, we can implement a fat client GUI. In this case, we can use:

- JavaBeans\(^5\) to implement the model and,
- SWT or Java Swings\(^6\) 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\(^7\) to implement the model,
- JSP\(^8\) to implement the view and,
- EJB session\(^9\) to implement the controller.

---

\(^3\) https://en.wikipedia.org/wiki/salt\%20\&\%28cryptography\%29
\(^4\) https://en.wikipedia.org/wiki/cryptographic\%20hash\%20function
\(^5\) https://en.wikibooks.org/wiki/Java\%20Programming\%2FJavaBeans
\(^6\) https://en.wikibooks.org/wiki/Java\%20Swings
\(^7\) https://en.wikibooks.org/wiki/Jakarta\%20EE\%20Programming\%2FJakarta\%20Enterprise\%20Beans
\(^8\) https://en.wikibooks.org/wiki/Jakarta\%20EE\%20Programming\%2FJakarta\%20Server\%20Pages
\(^9\) https://en.wikibooks.org/wiki/Jakarta\%20EE\%20Programming\%2FJakarta\%20Enterprise\%20Beans
17 Observer

17.1 Scope
Object

17.2 Purpose
Behavioral

17.3 Intent
Define a one-to-many dependency between objects so that when one object changes state (the Subject), all its dependents (the Observers) are notified and updated automatically.

17.4 Applicability
- when an abstraction has two aspects, one dependent on the other
- when a change to one object requires changing others, and you don't know how many objects need to be changed
- when an object should notify other objects without making assumptions about who these objects are

17.5 Structure

Figure 20
17.6 Consequences

- + modularity: subject and observers may vary independently
- + extensibility: can define and add any number of observers
- + customizability: different observers provide different views of subject
- - unexpected updates: observers don’t know about each other
- - update overhead: might need hints

17.7 Implementation

- subject-observer mapping
- dangling references
- avoiding observer-specific update protocols: the push and pull models
- registering modifications of interest explicitly

17.8 Related pattern

- Singleton\(^1\), is used to make observable object unique and accessible globally.
- Mediator\(^2\), is used to encapsulate updated objects

17.9 Description

**Problem**

In one place or many places in the application we need to be aware about a system event or an application state change. We’d like to have a standard way of subscribing to listening for system events and a standard way of notifying the interested parties. The notification should be automated after an interested party subscribed to the system event or application state change. There should be a way to unsubscribe, too.

**Forces**

Observers and observables probably should be represented by objects. The observer objects will be notified by the observable objects.

**Solution**

After subscribing the listening objects will be notified by a way of method call.

**Loose coupling**

When two objects are loosely coupled, they can interact, but they have very little knowledge of each other. Strive for loosely coupled designs between objects that interact.

- The only thing that Subject knows about an observer is that it implements a certain interface

---

\(^1\) Chapter 20 on page 251
\(^2\) Chapter 14 on page 189
• We can add new observers at any time
• We never need to modify the subject to add new types of observers
• We can reuse subjects or observers independently of each other
• Changes to either the subject or an observer will not affect the other

17.10 Examples

The Observer pattern is used extensively in Java. E.g. in the following piece of code

• button is the Subject
• MyListener is the Observer
• actionPerformed() is the equivalent to update()

```java
1 JButton button = new JButton("Click me!"),
2 button.addActionListener(new MyListener());
3 class MyListener implements ActionListener {
4     public void actionPerformed(ActionEvent event) {
5         ...
6     }
7 }
```

Another example is the PropertyChangeSupport. The Component class uses a PropertyChangeSupport object to let interested observers register for notification of changes in the properties of labels, panels, and other GUI components.

![Diagram](image)

Figure 21

Can you find the Subject, Observer and update() in the above class diagram?

• Component is the Subject
• PropertyChangeListener is the Observer
• propertyChange() is the equivalent to update()

17.11 Cost

This pattern can be tricky if you do not beware. Beware the data representing the state changes of the subject can evolve without changing the interfaces. If you only transmit a string, the pattern could become very expensive if at least one new observer needs also a
state code. You should use a mediator unless you know the implementation of the subject state will never change.

17.11.1 Creation
This pattern has a cost to create.

17.11.2 Maintenance
This pattern can be expensive to maintain.

17.11.3 Removal
This pattern has a cost to remove too.

17.12 Advises

- Put the subject and observer term in the name of the subject and the observer classes to indicate the use of the pattern to the other developers.
- To improve performances, you can only send to the observers the difference of states instead of the new state. The observers can only update following the changed part of the subject instead of all the state of the subject.

17.13 Implementations

```java
// 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 {

17.14 Traditional Method

C# and the other .NET Framework\(^3\) 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.

```csharp
using System;
using System.Collections;

namespace Wikipedia.Patterns.Observer
{
    // 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);
            }
        }
    }
}
```

\(^3\) https://en.wikibooks.org/wiki/.NET%20Framework
Using Events

The alternative to using concrete and abstract observers and publishers in C# and other .NET Framework\(^4\) languages, such as Visual Basic\(^5\), is to use events. The event model is supported via delegates\(^6\) 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.


\(^5\) https://en.wikibooks.org/wiki/Visual%20Basic

\(^6\) https://en.wikibooks.org/wiki/Delegation%20%28programming%29
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:
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.
    }
}

17.16 Handy implementation

You can implement this pattern in Java\(^7\) like this:

```java
// Observer pattern -- Structural example
@since JDK 5.0
import java.util.ArrayList;
...
```

\(^7\) https://en.wikibooks.org/wiki/Java\%20Programming
abstract class Subject {
    private ArrayList<Observer> observers;
    public void attach(Observer observer) {
        observers.add(observer);
    }
    public void detach(Observer observer) {
        observers.remove(observer);
    }
    public void notifyObservers() {
        for (Observer o : observers)
            o.update();
    }
}

class ConcreteSubject extends Subject {
    private String subjectState;
    public String getSubjectState() {
        return subjectState;
    }
    public void setSubjectState(String value) {
        subjectState = value;
    }
}

abstract class Observer {
    abstract public void update();
}

class ConcreteObserver extends Observer {
    private String name;
    private String observerState;
    private ConcreteSubject subject;
    public ConcreteObserver(ConcreteSubject subject, String name) {
        this.subject = subject;
        this.name = name;
        //subject.attach(this);
    }
    public void update() {
        observerState = subject.getSubjectState();
        System.out.printf("Observer %s's new state is %s\n", name, observerState);
    }
}

public class Client {
    public static void main(String[] args) {
        // Configure Observer structure
        ConcreteSubject s = new ConcreteSubject();
    }
}
s.attach(new ConcreteObserver(s, "A");
 s.attach(new ConcreteObserver(s, "B");
 s.attach(new ConcreteObserver(s, "C");

// Change subject and notify observers
s.setSubjectState("NEW");
s.notifyObservers();
}
}

17.17 Built-in support

The Java JDK has several implementations of this pattern: application in Graphical User Interfaces such as in the AWT toolkit, Swing etc. In Swing, whenever a user clicks a button or adjusts a slider, many objects in the application may need to react to the change. Swing refers to interested clients (observers) as "listeners" and lets you register as many listeners as you like to be notified of a component's events.

MVC is more M(VC) in Swing, i.e. View and Controller are tightly coupled; Swing does not divide Views from Controllers. MVC supports n-tier development, i.e. loosely coupled layers (see below) that can change independently and that may even execute on different machines.

There is also a built-in support for the Observer pattern. All one has to do is extend java.util.Observable\(^8\) (the Subject) and tell it when to notify the java.util.Observer\(^9\) s. The API does the rest for you. You may use either push or pull style of updating your observers.

\(^8\) [http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html](http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html)

\(^9\) [http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html](http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html)
java.util.Observable is a class while java.util.Observer is an interface.

```java
def set_value(double value) {
    this.value = value;
    set_changed();
    notify_observers();
}
```

Note that you have to call `setChanged()` so that the `Observable` code will broadcast the change. The `notifyObservers()` method calls the `update()` method of each registered observer. The `update()` method is a requirement for implementers of the `Observer` Interface.

```java
import java.util.Observable;
import java.util.Observer;

class ConcreteSubject extends Observable {
    // Fields
    private String subjectState;
    // Methods
    public void dataChanged() {
        setChanged();
        notifyObservers();
    }
    // Properties
    public String getSubjectState() {
```

Figure 22
16     return subjectState;
17 }  
18 public void setSubjectState(String value) {
19     subjectState = value;
20     dataChanged();
21 }  
22 }

1 // "ConcreteObserver"
2 import java.util.Observable;
3 import java.util.Observer;
4
5 class ConcreteObserver extends Observer {
6     // Fields
7     private String name;
8     private String observerState;
9     private Observable subject;
10
11     // Constructors
12     public ConcreteObserver(Observable subject, String name) {
13         this.subject = subject;
14         this.name = name;
15         subject.addObserver(this);
16     }
17
18     // Methods
19     public void update(Observable subject, Object arg) {
20         if (subject instanceof ConcreteSubject) {
21             ConcreteSubject subj = (ConcreteSubject)subject;
22             observerState = subj.getSubjectState();
23             System.out.printf("Observer %s's new state is %s\n", name,
24                 observerState);
25         }
26     }
27 }
28 }

1 // Client test
2 public class Client {
3     public static void main(String[] args) {
4         // Configure Observer structure
5         ConcreteSubject s = new ConcreteSubject();
6         new ConcreteObserver(s, "A");
7         new ConcreteObserver(s, "B");
8         new ConcreteObserver(s, "C");
9         // Change subject and notify observers
10         s.setSubjectState("NEW");
11     }
12 }

17.17.1 Keyboard handling

Below is an example written in Java that takes keyboard input and treats each input line as an event. The example is built upon the library classes java.util.Observer and java.util.Observable. When a string is supplied from System.in, the method 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.
```java
1 /* Filename : EventSource.java */
2 package org.wikibooks.obs;
3
4 import java.util.Observable; // Observable is here
5 import java.io.BufferedReader;
6 import java.io.IOException;
7 import java.io.InputStreamReader;
8
9 public class EventSource extends Observable implements Runnable {
10    @Override
11    public void run() {
12        try {
13            final InputStreamReader isr = new InputStreamReader(System.in);
14            final BufferedReader br = new BufferedReader(isr);
15            while (true) {
16                String response = br.readLine();
17                setChanged();
18                notifyObservers(response);
19            }
20        } catch (IOException e) {
21            e.printStackTrace();
22        }
23    }
24}
```

```java
1 /* Filename : ResponseHandler.java */
2 package org.wikibooks.obs;
3
4 import java.util.Observable;
5 import java.util.Observer; /* this is Event Handler */
6
7 public class ResponseHandler implements Observer {
8    private String resp;
9    public void update(Observable obj, Object arg) {
10        if (arg instanceof String) {
11            resp = (String) arg;
12            System.out.println("Received Response: "+ resp);
13        }
14    }
15}
```

```java
1 /* Filename : MyApp.java */
2 /* This is the main program */
3 package org.wikibooks.obs;
4
5 public class MyApp {
6    public static void main(String[] args) {
7        System.out.println("Enter Text >");
8
9        // create an event source - reads from stdin
10        final EventSource eventSource = new EventSource();
11
12        // create an observer
13        final ResponseHandler responseHandler = new ResponseHandler();
14
15        // subscribe the observer to the event source
16        eventSource.addObserver(responseHandler);
17
18        // starts the event thread
19        Thread thread = new Thread(eventSource);
20        thread.start();
21    }
```

The Java implementation of the Observer pattern has pros and cons:

**Pros**

- It hides many of the details of the Observer pattern
- It can be used both pull and push ways.

**Cons**

- Because `Observable` is a class, you have to subclass it; you can’t add on the `Observable` behavior to an existing class that subclasses another superclass (fails the programming to interfaces principle). If you can’t subclass `Observable`, then use delegation, i.e. provide your class with an Observable object and have your class forward key method calls to it.
- Because `setChanged()` is protected, you can’t favour composition over inheritance.

class STUDENT

```php
<?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());
    }
}
```
<?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";
    }
}
Observer

/* 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()." ".$classes->GET_name()." was removed";
        }
    }
}
/* Notify an observer */
public function notify(){
    foreach ($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>
</pre>
</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
}

The observer pattern in Python:

```python
class AbstractSubject:
    def register(self, listener):
        raise NotImplementedError("Must subclass me")
    def unregister(self, listener):
```

10 https://en.wikibooks.org/wiki/Python%20Programming
raise NotImplementedError("Must subclass me")

def notify_listeners(self, event):
    raise NotImplementedError("Must subclass me")

class Listener:
    def __init__(self, name, subject):
        self.name = name
        subject.register(self)

    def notify(self, event):
        print self.name, "received event", event

class Subject(AbstractSubject):
    def __init__(self):
        self.listeners = []
        self.data = None

    def getUserAction(self):
        self.data = raw_input('Enter something to do: ')
        return self.data

    # Implement abstract Class AbstractSubject
    def register(self, listener):
        self.listeners.append(listener)

    def unregister(self, listener):
        self.listeners.remove(listener)

    def notify_listeners(self, event):
        for listener in self.listeners:
            listener.notify(event)

if __name__=="__main__":
    # make a subject object to spy on
    subject = Subject()

    # register two listeners to monitor it.
    listenerA = Listener("<listener A>", subject)
    listenerB = Listener("<listener B>", subject)

    # simulated event
    subject.notify_listeners("<event 1>")
    # outputs:
    #  <listener A> received event <event 1>
    #  <listener B> received event <event 1>

    action = subject.getUserAction()
    subject.notify_listeners(action)
    #Enter something to do:hello
    # outputs:
    #  <listener A> received event hello
    #  <listener B> received event hello

The observer pattern can be implemented more succinctly in Python using function decorators\(^1\).

In Ruby, use the standard Observable mixin. For documentation and an example, see http://www.ruby-doc.org/stdlib/libdoc/observer/rdoc/index.html

\(^1\) https://en.wikibooks.org/wiki/Python%20syntax%20and%20semantics%23Decorators
18 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.

18.1 Structure

![UML class diagram describing the Prototype design pattern](image)

Figure 23  UML class diagram describing the Prototype design pattern

18.2 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
Abstract Factory, Builder, and Prototype can use Singleton in their implementations. (GoF, p126). 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.

18.3 Advices

- Put the prototype term in the name of the prototype classes to indicate the use of the pattern to the other developers.
- Only use an interface when it is necessary.

18.4 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.

The concrete type of object is created from its prototype. MemberwiseClone is used in the Clone method to create and return a copy of ConcreteFoo1 or ConcreteFoo2.

```csharp
public abstract class Foo
{
    // normal implementation

    public abstract Foo Clone();
}

public class ConcreteFoo1 : Foo
```
Implementations

```
public override Foo Clone()
{
    return (Foo)this.MemberwiseClone(); // Clones the concrete class.
}

public class ConcreteFoo2 : Foo
{
    public override Foo Clone()
    {
        return (Foo)this.MemberwiseClone(); // Clones the concrete class.
    }
}
```

Another example:

```
//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**
#include <iostream>

using namespace std;

// Prototype
class Prototype
{
public:
    virtual ~Prototype() {} 
    virtual Prototype* clone() const = 0;
    virtual void setX(int x) = 0;
    virtual int getX() const = 0;
    virtual void printX() 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
int main()
{
    Prototype* prototype = new ConcretePrototype(1000);
    for (int i = 1; i < 10; i++) {
        Prototype* temporaryPrototype = prototype->clone();
        temporaryPrototype->setX(temporaryPrototype->getX() * i);
        temporaryPrototype->printX();
        delete temporaryPrototype;
    }
    delete prototype;
    return 0;
}
This pattern creates the kind of object using its prototype. In other words, while creating the object of Prototype object, the class creates a clone of it and returns it as prototype. The clone method has been used to clone the prototype when required.

```java
// Prototype pattern
public abstract class Prototype implements Cloneable {
    public Prototype clone() throws CloneNotSupportedException{
        return (Prototype) super.clone();
    }
}
public class ConcretePrototype1 extends Prototype {
    @Override
    public Prototype clone() throws CloneNotSupportedException {
        return (ConcretePrototype1)super.clone();
    }
}
public class ConcretePrototype2 extends Prototype {
    @Override
    public Prototype clone() throws CloneNotSupportedException {
        return (ConcretePrototype2)super.clone();
    }
}

Another example:

```java
1 /**
2 * Prototype class
3 */
4 interface Prototype extends Cloneable {
5     void setX(int x);
6     void printX();
7     int getX();
8 }
9}
10 */

1 /**
2 * Implementation of prototype class
3 */
4 class PrototypeImpl implements Prototype {
5     private int x;
6     /**
7     * Constructor
8     */
9     public PrototypeImpl(int x) {
10         setX(x);
11     }
12 }
```
```java
abstract class Prototype
{
    public string $a;
    public string $b;

    @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();
    }
}

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
```

// The Prototype pattern in PHP is done with the use of built-in PHP function __clone()
public function displayCONS(): void
{
    echo "CONS: {$this->a}\n";
    echo "CONS: {$this->b}\n";
}

public function displayCLON(): void
{
    echo "CLON: {$this->a}\n";
    echo "CLON: {$this->b}\n";
}

abstract function __clone();
}

class ConcretePrototype1 extends Prototype
{
    public function __construct()
    {
        $this->a = "A1";
        $this->b = "B1";
        $this->displayCONS();
    }

    function __clone()
    {
        $this->displayCLON();
    }
}

class ConcretePrototype2 extends Prototype
{
    public function __construct()
    {
        $this->a = "A2";
        $this->b = "B2";
        $this->displayCONS();
    }

    function __clone()
    {
        $this->a = $this->a . "-C";
        $this->b = $this->b . "-C";
        $this->displayCLON();
    }
}
$cP1 = new ConcretePrototype1();
$cP2 = new ConcretePrototype2();
cP2C = clone $cP2;
// RESULT: #quanton81

// CONS: A1
// CONS: B1
// CONS: A2
// CONS: B2
// CLON: A2-C
// CLON: B2-C

Another example:
Let's write an occurrence browser class for a text. This class lists the occurrences of a word in a text. Such an object is expensive to create as the locations of the occurrences need an expensive process to find. So, to duplicate such an object, we use the prototype pattern:

```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.
         */
        // $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
*/
```

Let's write an occurrence browser class for a text. This class lists the occurrences of a word in a text. Such an object is expensive to create as the locations of the occurrences need an expensive process to find. So, to duplicate such an object, we use the prototype pattern:

```ruby
class WordOccurrences is
    field occurrences is
        The list of the index of each occurrence of the word in the text.

    constructor WordOccurrences(text, word) is
```
input: the text in which the occurrences have to be found
input: the word that should appear in the text
Empty the occurrences list
for each
textIndex
in text

isMatching

g := true
for each
wordIndex
in word
if the current word character does not match the current text character then

isMatching

g := false
if
isMatching
is true then
Add the current
textIndex
into the occurrences list

method getOneOccurrenceIndex(n) is
input: a number to point on the nth occurrence.
output: the index of the nth occurrence.
Return the nth item of the occurrences field if any.

method clone() is
output: a WordOccurrences object containing the same data.
Call clone() on the super class.
On the returned object, set the occurrences field with the value of the local occurrences field.
Return the cloned object.

An example of using this implementation:
text := "The prototype pattern is a creational design pattern in software development first described in design patterns, the book."
word := "pattern"
searchEngine := new WordOccurrences(text, word)
anotherSearchEngine := searchEngine.clone()
Prototype

```python
# Call _copy_func to ensure the attributes of the objects are identical
self._copy_func(self.instance, clone)
return clone

def _copy_func(self, fromObj, toObj):
    # Dual purpose function which is assigned to the decorated object
    # and used internally in the decorator to copy the original attributes
    # to the clone to ensure it's identical to the object which made the
    clone
    for attr in dir(fromObj):
        setattr(toObj, attr, getattr(fromObj, attr))

def __init__(self, cls):
    # This is only called once per decorated class type so self.cls
    # should remember the class that called it so it can generate
    # unique objects of that class later on (in __call__)
    self.cls = cls

    # NOTE: on a decorator "__call__" MUST be implemented
    # this function will automatically be called each time a decorated
    # object is cloned/created
    def __call__(self):
        # Create an instance of the class here so a unique object can be
        # sent back to the constructor
        self.instance = self.cls()
        # Assign the private functions back to the unique class
        # (which is the whole point of this decorator)
        self.instance.Clone = self._clone_func
        self.instance.Copy = self._copy_func
        # Finally, send the unique object back to the object's constructor
        return self.instance

@Prototype
class Concrete:
    def __init__(self):
        # Test value to see if objects are independently generated as
        # well as if they properly copy from one another
        self.somevar = 0

@Prototype
class another:
    def __init__(self):
        self.somevar = 50

if __name__ == '__main__':
    print "Creating A"
    a = Concrete()
    print "Cloning A to B"
    b = a.Clone()
    print "A and B somevars"
    print a.somevar
    print b.somevar
    print "Changing A somevar to 10..."
    a.somevar = 10
    print "A and B somevars"
    print a.somevar
    print b.somevar
    print "Creating another kind of class as C"
    c = another()
    print "Cloning C to D"
    d = c.Clone()
    print "Changing C somevar to 100"
    c.somevar = 100
    print "C and D somevars"
```

236
Implementation:

```python
print c.somevar
print d.somevar
```

Output:

```
Creating A
Cloning A to B
A and B somevars
0
0
Changing A somevar to 10...
A and B somevars
10
0
Creating another kind of class as C
Cloning C to D
Changing C somevar to 100
C and D somevars
100
50
```
19 Proxy

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.

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

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

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

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

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

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

    public Driver(int age)
    {
        this.Age = age;
    }
}
```
// How to use above Proxy class?
private void btnProxy_Click(object sender, EventArgs e)
{
    ICar car = new ProxyCar(new Driver(15));
    car.DriveCar();
    car = new ProxyCar(new Driver(25));
    car.DriveCar();
}

Output

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

Notes:

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

Another example:

```csharp
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
        {
```
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
```

```
#include <iostream>
#include <memory>

class ICar {
public:
  virtual ~ICar() { std::cout << "ICar destructor!\n"; }

  virtual void DriveCar() = 0;
};

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

class ProxyCar : public ICar {
public:
  ProxyCar(int driver_age) : driver_age_(driver_age) {}

  void DriveCar() override {
    if (driver_age_ > 16) {
      real_car_->_DriveCar();
    }
  }

private:
  int driver_age_;
  std::shared_ptr<ICar> real_car_;  // proxy to the real car
};
```
abstract class AbstractCar
    abstract def drive
end

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

class Driver
    getter age : Int32
    def initialize(@age)
    end
end

class ProxyCar < AbstractCar
    private getter driver : Driver
    private getter real_car : AbstractCar
    def initialize(@driver)
        @real_car = Car.new
    end
    def drive
        if driver.age <= 16
            puts "Sorry, the driver is too young to drive."
        else
            @real_car.drive
        end
    end
end

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

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

Output

242
// Proxy Design pattern
unit DesignPattern.Proxy;

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

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

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

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

  // Proxy Object
  TProxyCar = Class(TInterfacedObject, ICar)
  private
    FDriver: IDriver;
    FRealCar: ICar;
  public
    constructor Create(Driver: IDriver); Overload;
    class function New(Driver: IDriver): ICar;
    procedure DriveCar;
  End;

implementation

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

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

{ TDriver Implementation }
constructor TDriver.Create(Age: Integer);
begin
  inherited Create;
end;
```
FAge := Age;
end;

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

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

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

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

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

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

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

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

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

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

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

// On System A
class RealImage implements Image {
    private final String filename;

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

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

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

// On System B
class ProxyImage implements Image {
    private final String filename;
    private RealImage image;

    /**
     * Constructor
     * @param filename
     */
    public ProxyImage(String filename) {
        this.filename = filename;
    }

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

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

Output

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

// Driver class
class Driver {
    constructor (age) {
        this.age = age
    }
}

// Car class
class Car {
    drive () {
        console.log('Car has been driven!')
    }
}

// Proxy car class
class ProxyCar {
    constructor (driver) {
        this.car = new Car()
        this.driver = driver
    }

drive () {
    if (this.driver.age <= 16) {
        console.log('Sorry, the driver is too young to drive.')
    } else {
        this.car.drive()
    }
}

// Run program
const driver = new Driver(16)
const car = new ProxyCar(driver)
car.drive()

const driver2 = new Driver(25)
const car2 = new ProxyCar(driver2)
car2.drive()}
Implementations

Output

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

More advanced proxies involve the **Proxy** object, which can intercept and redefine fundamental operations such as accessing properties. The handler functions in this case are sometimes called *traps*.¹

```
<?php

interface Image
{
    public function displayImage();
}

// On System A
class RealImage implements Image
{
    private string $filename = null;
    public function __construct(string $filename)
    {
        $this->filename = $filename;
        $this->loadImageFromDisk();
    }

    /**
     * Loads the image from the disk
     */
    private function loadImageFromDisk()
    {
        echo "Loading {$this->filename}" . PHP_EOL;
    }

    /**
     * Displays the image
     */
    public function displayImage()
    {
        echo "Displaying {$this->filename}" . PHP_EOL;
    }
}

// On System B
class ProxyImage implements Image
{
    private ?Image $image = null;
    private string $filename = null;
    public function __construct(string $filename)
    {
        $this->filename = $filename;
    }

    /**
     * Displays the image
     */
    public function displayImage()
    {
        if ($this->image === null) {
            echo "Proxy Image: Image not available" . PHP_EOL;
        } else {
            echo "Proxy Image: Image displayed" . PHP_EOL;
        }
    }
}
```

¹ Proxy - JavaScript | MDN². Retrieved

247
Proxy

```php
$this->image = new RealImage($this->filename);

$this->image->displayImage();

$image1 = new ProxyImage("HiRes_10MB_Photo1");
$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
```

Output

```
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
```

"""
Proxy pattern example.
""
```
from abc import ABCMeta, abstractmethod

NOT_IMPLEMENTED = "You should implement this."

class AbstractCar:
    __metaclass__ = ABCMeta

    @abstractmethod
def drive(self):
        raise NotImplementedError(NOT_IMPLEMENTED)

class Car(AbstractCar):
def drive(self) -> None:
    print("Car has been driven!")

class Driver:
def __init__(self, age: int) -> None:
    self.age = age

class ProxyCar(AbstractCar):
def __init__(self, driver) -> None:
    self.car = Car()
    self.driver = driver
def drive(self) -> None:
    if self.driver.age <= 16:
        print("Sorry, the driver is too young to drive.")
    else:
```
trait ICar {
    fn drive(&self);
}

struct Car {}

impl ICar for Car {
    fn drive(&self) {
        println!("Car has been driven!");
    }
}

impl Car {
    fn new() -> Car {
        Car {}
    }
}

struct ProxyCar<'a> {
    real_car: &'a ICar,
    driver_age: i32,
}

impl<'a> ICar for ProxyCar<'a> {
    fn drive(&self) {
        if self.driver_age > 16 {
            self.real_car.drive();
        } else {
            println!("Sorry, the driver is too young to drive.");
        }
    }
}

impl<'a> ProxyCar<'a> {
    fn new(driver_age: i32, other_car: &'a ICar) -> ProxyCar {
        ProxyCar {
            real_car: other_car,
            driver_age: driver_age,
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
fn test_underage() {
    let car = Car::new();
    let proxy_car = ProxyCar::new(16, &car);
    proxy_car.drive();
}

#[test]
fn test_can_drive() {
    let car = Car::new();
    let proxy_car = ProxyCar::new(17, &car);
    proxy_car.drive();
}

Output

Sorry, the car is too young for you to drive.
Car has been driven!
20 Singleton

The term **Singleton** refers to an object that can be instantiated only once. Use the Singleton Design pattern if:

- You need global access to a resource, like logging ...
- You need only one instance of a utility class, do not want to create lots of objects.

In some applications, it is appropriate to enforce a single instance of an object, for example: window managers, print spoolers, database access, and filesystems.

20.1 Refresh Java object creation

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:

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

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

```java
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:

```java
Animal dog = new Animal();
Animal cat = 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 effect 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:

```java
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 cannot be instantiated yet you can obtain an instance of the object type. Let us create a normal object using Java.

```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.

```java
1 class TestHarness {
2     public static void main(String[] args) {
3         NormalObject object = new NormalObject();
4     }
5 }
```

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 paranthesis and into another class definition because it is declared a public accessor while creating the Constructor in the above example.

### Creating singleton object

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**.

```java
1 class SingletonObject {
2     private SingletonObject() {
3     }
4 }
```

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.

```java
1 class TestHarness {
2     public static void main(String[] args) {
3         SingletonObject object = new SingletonObject();
4     }
5 }
```

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.

```java
1 class SingletonObject {
2     private static SingletonObject object;
3     
4     private SingletonObject() {
5         // Instantiate the object.
6     }
7     
8     public static SingletonObject getInstance() {
9         if (object == null) {
10             object = new SingletonObject(); // Create the object for the first
11             and last time
12         }
13         return object;
14     }
15 }
```

The changes involve adding a static class field 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:
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 getting multiple objects and see what happens.

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:

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 cannot be instantiated more than once. Typically it is obtained using a static custom implementation.

20.2 Singleton & Multi Threading

Java uses multi threading 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 singelton thread safe, we have three options: 1. Synchronize the method getInstance(), which would look like:

Synchronizing the method guarantees that the method will never be called twice at the same time. The problem with the above code is that 'synchronization' is expensive. 'Synchronization' checks will occur every time the function is called. Therefore, the above code should not be the first option. 2. Another approach would be to create a singleton instance as shown below:
The above code will be much faster; once the singleton instance is created no synchronization is needed. It just returns the same instance reference. 3. Use static initialization block to create the instance. The JVM makes sure that the static block is executed once when the class is loaded.

20.3 Examples

In Java, the class `java.lang.Runtime` is a singleton. Its constructor is protected and you can get an instance by calling the `getRuntime()` method.

20.4 Cost

Beware of this design pattern! It creates the same issues than the global variables in procedural programming so it can be hard to debug.

20.4.1 Creation

It can be expensive. You may have to refactor all the instantiations of the class, unless the class is new.
20.4.2 Maintenance

There is no additional cost.

20.4.3 Removal

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

20.5 Advises

- Name the method `getInstance()` to indicate the use of the pattern to the other developers.
- Use this design pattern for frozen data like configuration or external data. You will not have debugging issues.

20.6 Implementation

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".

```scala
object MySingleton {
    println("Creating the singleton")
    val i : Int = 0
}
```

20.7 Traditional simple way using synchronization

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

```java
public class Singleton {
    private volatile static Singleton singleton; // volatile is needed so that multiple thread can reconcile the instance
    private Singleton(){}  
    public static SingletongetSingleton() { // synchronized keyword has been removed from here
        if (singleton == null) { // needed because once there is singleton available no need to acquire 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;
    }
}
```
20.8 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.

```java
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;
    }
}
```

20.9 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.

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

Singleton pattern in D programming language

```d
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() {
        writeln("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();
    }

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() {
        ...
    }
}
Singleton

Singleton::getInstance()->doAction();
?

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.

```java
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;
        }
    }
}
```

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

```objective-c
@interface MySingleton : NSObject
{

}
@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:

```objective-c
+ (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.
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. The simplest of all is:

```objc
@interface MySingleton : NSObject
{
}
@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
```

This example is thread-safe with lazy initialization.

```objc
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()
    {
    }
}
```

```csharp
public class Singleton
{
    // Use Lazy<T> to lazily initialize the class and provide thread-safe access
    private static readonly Lazy<Singleton> _lazyInstance = new Lazy<Singleton>(() => new Singleton());
    public static Singleton Instance
    {
        // Class implements singleton pattern.
    }
}
```
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#: [http://www.c-sharpcorner.com/UploadFile/snorrebaard/GenericSingleton11172008110419AM/GenericSingleton.aspx](http://www.c-sharpcorner.com/UploadFile/snorrebaard/GenericSingleton11172008110419AM/GenericSingleton.aspx)

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:

```delphi
unit Singleton;
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)
private
  protected SingletonCreator() {
    protected Singleton() { }
  }
  // explicit static constructor to disable beforefieldinit
  static SingletonCreator() { }
  public static S CreatorInstance
  {
    get { return instance; }
  }
  public static T Instance
  {
    get { return SingletonCreator<T>.CreatorInstance; }
  }
end;

// Concrete Singleton
public class TestClass : Singleton<TestClass>
{
  public string TestProc()
  {
    return "Hello World";
  }
}
// Somewhere in the code
......
TestClass.Instance.TestProc();
......
```
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.

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:

```python
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:

```python
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`.

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

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

```python
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.

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

262
# 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

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\(^1\) extension module. 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

\(^1\) [http://search.cpan.org/perldoc?MooseX::Singleton](http://search.cpan.org/perldoc?MooseX::Singleton)
In ABAP Objects, to make instantiation private, add an attribute of type ref to the class, and a static method to control instantiation.

```abap
program pattern_singleton.

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

class lcl_Singleton definition create private.

public section.

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

private section.

class-data:
  fg_Singleton type ref to lcl_Singleton.

endclass.

class lcl_Singleton implementation.

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

endclass.
```
21 State

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

```cpp
#include <iostream>
using namespace std;
class Machine {
  class State *current;
  public:
    Machine();
    void setCurrent(State *s)
    {
      current = s;
    }
    void on();
    void off();
};
class State {
  public:
    virtual void on(Machine *m)
    {
      cout << " already ON\n";
    }
    virtual void off(Machine *m)
    {
      cout << " already OFF\n";
    }
};
void Machine::on()
{
  current->on(this);
}
void Machine::off()
{
  current->off(this);
}
class ON: public State {
  public:
    ON() {
      cout << " ON-ctor ";
    }
    "ON() {
    cout << " dtor-ON\n";
    void off(Machine *m);
  };
class OFF: public State
```
{  
  public:
  
  OFF()
  {
    cout << " OFF-ctor ";
  };

  ~OFF()
  {
    cout << " dtor-OFF\n";
  };
  
  void on(Machine *m)
  {
    cout << " going from OFF to ON";
    m->setCurrent(new ON());
    delete this;
  }

  void OFF::off(Machine *m)
  {
    cout << " going from ON to OFF";
    m->setCurrent(new OFF());
    delete this;
  }

  Machine::Machine()
  {
    current = new OFF();
    cout << '\n';
  }

  int main()
  {
    void(Machine:: *ptrs[])() =
    {
      Machine::off, Machine::on
    };
    Machine fsm;
    int num;
    while (1)
    {
      cout << "Enter 0/1: ";
      cin >> num;
      (fsm. *ptrs[num])();
    }
  
  
}

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();
  }
}
The state interface and two implementations. The state's method has a reference to the context object and is able to change its state.

```csharp
public abstract void Handle(Context context);
```

```csharp
context.State = new ConcreteStateB();
```

```csharp
context.State = new ConcreteStateA();
```

```csharp
public void Request()
{
    state.Handle(this);
}
```
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.
The Enum-way

The test below shows also the usage:

```java
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
```

```
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)} = 
        ([1260,1380,1510], 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{qw('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();

import itertools

270
"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

1 revised from http://ginstrom.com/scribbles/2007/10/08/design-patterns-python-style/
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
The state interface and two implementations. The state's method has a reference to the context object and is able to change its state.

```php
interface IState {
    public function write(StateContext $context, string $name): void;
}
```

```php
definal class LowerCaseState implements IState {
    public function write(StateContext $context, string $name): void {
        print Str\lowercase($name) . "\n";
        $context->setState(new MultipleUpperCaseState());
    }
}
```

```php
definal class MultipleUpperCaseState implements IState {
    private int $count = 0;
    public function write(StateContext $context, string $name): void {
        print Str\uppercase($name) . "\n";
        ++$this->count;
        if ($this->count > 1) {
            $context->setState(new LowerCaseState());
        }
    }
}
```

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.

```php
definal class StateContext {
    public function __construct()
    {
        private IState $state = new LowerCaseState();
    }

    /**
     * Set the current state.
     * Normally only called by classes implementing the State interface.
     */
    public function setState(IState $state): void {
        $this->state = $state;
    }

    public function write(string $name): void {
        $this->state->write($this, $name);
    }
}
```
The test below shows also the usage:

```php
require 'vendor/hh_autoload.hh';

<<__EntryPoint>>
async function main(): Awaitable<void> {
    $context = new StateContext();
    $context->write('Monday');
    $context->write('Tuesday');
    $context->write('Wednesday');
    $context->write('Thursday');
    $context->write('Friday');
    $context->write('Saturday');
    $context->write('Sunday');
}
```

According to the above code, the output of main() entry point should be:

```
monday
TUESDAY
WEDNESDAY
thursday
FRIDAY
SATURDAY
sunday
```

### 21.1 References
22 Strategy

22.1 Scope
Object

22.2 Purpose
Behavioral

22.3 Intent
Define a family of algorithms, encapsulate each one, and make them interchangeable to let clients and algorithms vary independently from the clients using it.

22.4 Applicability
• when an object should be configurable with one of several algorithms,
• and all algorithms can be encapsulated,
• and one interface covers all encapsulations

22.5 Structure

Figure 24
22.6 Consequences

- + greater flexibility, reuse
- + can change algorithms dynamically
- - strategy creation & communication overhead
- - inflexible Strategy interface

22.7 Implementation

- exchanging information between a Strategy and its context
- static strategy selection via templates

22.8 Related patterns

- State\(^1\), can activate several states, whereas a strategy can only activate one of the algorithms.
- Flyweight\(^2\), provides a shared object that can be used in multiple contexts simultaneously, whereas a strategy focuses on one context.
- Decorator\(^3\), changes the skin of an object, whereas a strategy changes the guts of an object.
- Composite\(^4\), is used in combination with a strategy to improve efficiency.

22.9 Description

Suppose that you work for a company that builds a strategy game. Let's assume that you've come up with the following hierarchy of classes. All characters are able to walk, and there is also a method to render them on screen. The superclass takes care of the implementation of `walk()` method, while each subclass provides its own implementation of `display()` since each character looks different.
A new requirement arrives that the characters need to fight, too. Simple job you say; just add a `fight()` method to `Character` superclass. But, wait a moment; what about Workers? They cannot fight!

```java
1 class Worker {
2     ....
```

Well, one could think that you could simply override `fight()` method in `Worker` subclass to just do nothing.
But what if in the future there is a need for a `Guard` class that can fight but shouldn’t walk? We need a cleaner solution. What about taking out the fight and walk behaviors from the superclass to an interface? That way, only the characters that are supposed to walk should implement the `Walkable` interface and only those characters that are supposed to fight should implement the `Fightable` interface.

![Class diagram](image)

**Figure 27**

Well, this is another way to say for duplicate code. What if you need to make a small change to the fight behavior? You would need to modify all the classes. At this point we should put down three design principles to follow in our application development:

1. **Identify the aspects of your application that vary and separate them from what stays the same.** Take what varies and “encapsulate” it so it won’t affect the rest of your code.
2. **Program to an interface not to an implementation.**
3. *Favour composition over inheritance.*

So, let’s apply the 1st design principle. Pull out fight and walk behavior to different classes. And to apply the 2nd design principle as well, we have to pull out these behaviors to interfaces. Hence, we create a new WeaponBehavior interface to deal with fight behavior, and similarly a WalkBehavior interface to deal with walk behavior.

![Figure 28](image)

**Characters’** behaviors live in separate classes that implement a particular behavior interface. That way, the Character classes don’t need to know any of the implementation details for their own behaviors. In addition, we no more rely on an implementation but on an interface. Other types of objects can use these behaviors, too, because they’re not hidden inside our Character class. And we can add new behaviors without modifying any of the existing ones or touching our character classes. So now, all we have to do is have our Character class delegate all behavior information to the two behavior interfaces (so here comes the 3rd design principle). We do that by adding two instance variables to Character, weapon and walk.

![Figure 29](image)
public abstract class Character {
  private WeaponBehavior weapon;
  private WalkBehavior walk;
  public void fight() {
    weapon.useWeapon(); // delegation of fight behavior
  }
  public void setWeapon(WeaponBehavior w) {
    weapon = w;
  }
  abstract void display();
}

Each character object will set these variables polymorphically to reference the specific behavior type it would like at runtime.

public class Knight extends Character {
  public Knight() {
    weapon = new SwordBehavior();
    walk = new GallopBehavior();
  }
  public void display() {
    ...
  }
}

Think of these behaviors as families of algorithms. So, composition gives you a lot of flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime as long as the object you are composing with, implements the correct behavior interface.

### 22.10 Examples

#### 22.11 Cost

Think twice before implementing this pattern. You have to be sure your need is to frequently change an algorithm. You have to clearly anticipate the future, otherwise, this pattern will be more expensive than a basic implementation.

#### 22.11.1 Creation

This pattern is expensive to create.

#### 22.11.2 Maintenance

This pattern can be expensive to maintain. If the representation of a class often changes, you will have lots of refactoring.
### 22.11.3 Removal

This pattern is hard to remove too.

### 22.12 Advises

- Use the *strategy* term to indicate the use of the pattern to the other developers.

### 22.13 Implementations

The strategy pattern in ActionScript\(^5\) 3:

```plaintext
//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" );
        }
    }
}

A struct in C⁶ 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:

```c
#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" }; // function pointer
    Button button2 = { print_array, "List 'em" }; // function pointer

    int n = 10;
    int numbers[n];
    for (int i = 0; i < n; i++)
        numbers[i] = i;
```
The strategy pattern in C++\(^7\) is similar to Java, but does not require dynamic allocation of objects.

```cpp
#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)
    {
        std::cout << "Called ConcreteStrategyAdd's execute()\n";
        return a + b;
    }
};

class ConcreteStrategySubtract:public Strategy
{
public:
    int execute(int a, int b)
    {
        std::cout << "Called ConcreteStrategySubtract'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)
    {
    }
```

\(^7\) https://en.wikibooks.org/wiki/C%2B%2B%20Programming
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 << "resultB: " << resultB << "resultC: " << resultC << "\n";
}

public class StrategyPatternWiki
{
    public static void Main(String[] args)
    {
        // Prepare strategies
        var normalStrategy = new NormalStrategy();
        var happyHourStrategy = new HappyHourStrategy();
        var firstCustomer = new CustomerBill(normalStrategy);

        // Normal billing
        firstCustomer.Add(1.0, 1);

        // Start Happy Hour
        firstCustomer.Strategy = happyHourStrategy;
        firstCustomer.Add(1.0, 2);

        // New Customer
        var secondCustomer = new CustomerBill(happyHourStrategy);
        secondCustomer.Add(0.8, 1);

        // The Customer pays
        firstCustomer.Print();

        // End Happy Hour
        secondCustomer.Strategy = normalStrategy;
        secondCustomer.Add(1.3, 2);
        secondCustomer.Add(2.5, 1);
        secondCustomer.Print();
    }
}

// CustomerBill as class name since it narrowly pertains to a customer's bill
class CustomerBill
{
    private IList<double> drinks;
// Get/Set Strategy
public IBillingStrategy Strategy { get; set; }

public CustomerBill(IBillingStrategy strategy)
{
    this.drinks = new List<double>();
    this.Strategy = strategy;
}

class NormalStrategy : IBillingStrategy
{
    public double GetActPrice(double rawPrice) => rawPrice;
}

class HappyHourStrategy : IBillingStrategy
{
    public double GetActPrice(double rawPrice) => rawPrice * 0.5;
}

Another Example:

Delegates in C# 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.
```csharp
Func<int, int, int> concreteStrategy1 = PerformLogicalBitwiseOr;

// Anonymous Delegate
Func<int, int, int> concreteStrategy2 = delegate(int op1, int op2) {
    return op1 & op2;
};

// Lambda Expressions
Func<int, int, int> concreteStrategy3 = (op1, op2) => op1 >> op2;
Func<int, int, int> concreteStrategy4 = (op1, op2) => op1 << op2;

class Context<T>
{
    public Func<T, T, T> Strategy { get; set; }
    public T Execute(T operand1, T operand2)
    {
        return this.Strategy != null
            ? this.Strategy(operand1, operand2)
            : default(T);
    }
}

static int PerformLogicalBitwiseOr(int op1, int op2)
{
    return op1 | op2;
}

22.13.1 Using interfaces

using System;

namespace Wikipedia.Patterns.Strategy
{
    // 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)
            {
                // Additional logic here...
            }
        }
    }
}
```
Implementations

```csharp
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!
    }
}
```

An example in Common Lisp\(^9\): Using strategy classes:

```lisp
(defun class context ()
    ((strategy :initarg :strategy :accessor strategy)))
(defun method execute ((c context))

\(^9\) https://en.wikibooks.org/wiki/Common%20Lisp
```
In Common Lisp using first class functions:

```lisp
(defun context ()
  ((strategy :initarg :strategy :accessor strategy)))

(defun 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))
```

Similar to Python and Scala, Falcon supports first-class functions. The following implements the basic functionality seen in the Python example.

```falcon
/*#
@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"
```

10 https://en.wikibooks.org/wiki/Falcon%20Programming
Fortran\textsuperscript{11} 2003 adds procedure pointers, abstract interfaces and also first-class functions. The following mirrors the Python example.

```fortran
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)
  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
```

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.

\textsuperscript{11} https://en.wikibooks.org/wiki/Fortran
Strategy

```java
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
```

An example in Java:\(^\text{12}\):

```java
/** The classes that implement a concrete strategy should implement this. */
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);

        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);
    }
}

An example in Java:

/** Imports a type of lambdas taking two arguments of the same type T and returning one argument of same type T */
import java.util.function.BinaryOperator;

/** Implements and assigns to variables the lambdas to be used later in configuring Context. */
// FunctionalUtils is just a convenience class, as the code of a lambda might be passed directly to Context constructor, as for ResultD below, in main().
/**
class FunctionalUtils {
    static final BinaryOperator<Integer> add = (final Integer a, final Integer b) -> {
        System.out.println("Called add's apply().");
        return a + b;
    };

    static final BinaryOperator<Integer> subtract = (final Integer a, final Integer b) -> {
        System.out.println("Called subtract's apply().");
        return a - b;
    };

    static final BinaryOperator<Integer> multiply = (final Integer a, final Integer b) -> {
        System.out.println("Called multiply's apply().");
        return a * b;
    };
}

/** Configured with a lambda and maintains a reference to a lambda */
class Context {

...
/** a variable referencing a lambda taking two Integer arguments and returning an Integer: */
private final BinaryOperator<Integer> strategy;

public Context(final BinaryOperator<Integer> lambda) {
    strategy = lambda;
}

public int executeStrategy(final int a, final int b) {
    return strategy.apply(a, b);
}

/** Tests the pattern */
public class StrategyExample {

    public static void main(String[] args) {
        Context context;
        context = new Context(FunctionalUtils.add);
        final int resultA = context.executeStrategy(3,4);

        context = new Context(FunctionalUtils.subtract);
        final int resultB = context.executeStrategy(3, 4);

        context = new Context(FunctionalUtils.multiply);
        final int resultC = context.executeStrategy(3, 4);

        context = new Context((final Integer a, final Integer b) -> a * b + 1);
        final int resultD = context.executeStrategy(3,4);

        System.out.println("Result A: " + resultA );
        System.out.println("Result B: " + resultB );
        System.out.println("Result C: " + resultC );
        System.out.println("Result D: " + resultD );
    }
}

Similar to Python and Scala, JavaScript supports first-class functions. The following implements the basic functionality seen in the Python example.

```javascript
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");
```

Perl\textsuperscript{15} has first-class functions, so as with Python, JavaScript and Scala, this pattern can be implemented without defining explicit subclasses and interfaces:

\begin{verbatim}
  sort { lc($a) cmp lc($b) } @items
\end{verbatim}

The strategy pattern can be formally implemented with Moose\textsuperscript{16}:

\begin{verbatim}
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;

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;

\end{verbatim}

\textsuperscript{15}  https://en.wikibooks.org/wiki/Perl
\textsuperscript{16}  https://en.wikibooks.org/wiki/Moose%20%28Perl%29
The strategy pattern in PHP:

```php
<?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 {
    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();
?>
```

PowerShell\textsuperscript{18} 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.

```powershell
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

```powershell
Function Context ([scriptblock]$strategy){
    { & $strategy }.GetNewClosure()
}
$a = Context {'Style A'}
$a.Invoke()
& (Context {'Style B'})
$c = Context {'Style C'}
& $c
```

The following example is equivalent to the C\# example 1 above, but in Python.

```python
import abc

class Bill:
    def __init__(self, billing_strategy: "BillingStrategy"):  
        self.drinks: list[float] = []
        self.billing_strategy = billing_strategy

    def add(self, price: float, quantity: int) -> None:
        self.drinks.append(self.billing_strategy.get_act_price(price * quantity))

    def __str__(self) -> str:
        return f"£{sum(self.drinks)}"

class BillingStrategy(abc.ABC):
    @abc.abstractmethod
    def get_act_price(self, raw_price: float) -> float:
        raise NotImplementedError

class NormalStrategy(BillingStrategy):
    def get_act_price(self, raw_price: float) -> float:
        return raw_price

class HappyHourStrategy(BillingStrategy):
    def get_act_price(self, self, raw_price: float) -> float:
        return raw_price

```
def get_act_price(self, raw_price: float) -> float:
    return raw_price * 0.5

def main() -> None:
    normal_strategy = NormalStrategy()
    happy_hour_strategy = HappyHourStrategy()

    customer_1 = Bill(normal_strategy)
    customer_2 = Bill(normal_strategy)

    # Normal billing
    customer_1.add(2.50, 3)
    customer_1.add(2.50, 2)

    # Start happy hour
    customer_1.billing_strategy = happy_hour_strategy
    customer_2.billing_strategy = happy_hour_strategy
    customer_1.add(3.40, 6)
    customer_2.add(3.10, 2)

    # End happy hour
    customer_1.billing_strategy = normal_strategy
    customer_2.billing_strategy = normal_strategy
    customer_1.add(3.10, 6)
    customer_2.add(3.10, 2)

    # Print the bills;
    print(customer_1)
    print(customer_2)

if __name__ == '__main__':
    main()

Second example in Python\textsuperscript{19}:

class Strategy:
    def execute(self, 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

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

\textsuperscript{19} https://en.wikibooks.org/wiki/Python
Implementations

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:

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"

An example in Ruby:

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

22.13.2 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.

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

Like Python, Scala\textsuperscript{21} also supports first-class functions. The following implements the basic functionality shown in the Python example.

```
// 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
```

\textsuperscript{21} https://en.wikibooks.org/wiki/Scala
23 Template method

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.

```csharp
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()\n    ");
}

/**
 * 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;
}

/**
 * The same example from above but with Java 8
 * functional interfaces. Note that now Game
class doesn’t have to be abstract.
*/
class Game {
  protected int playersCount;

  /*
   * Note that abstract methods are now replaced
   * with functional interface instances. It gives you a lot
   * more of flexibility: you can require all of them to be set
   * via constructor, you can wrap them or you can set default
   * implementations.
   */
  private final Runnable initializeGame;
  private final Consumer<Integer> makePlay;
  private final Supplier<Boolean> endOfGame;
  private final Runnable printWinner;

  /* Constructor with parameters initialization.
   * Note that you can define multiple constructors
   * with different sets of parameters or define
   * default values for null values. Subclasses
   * are required to call at least one of constructors
   * if you don’t define a default one.
   */
  protected Game(Consumer<Integer> makePlay, Supplier<Boolean> endOfGame) {
    this.makePlay = Optional.ofNullable(makePlay).orElseGet(() -> {});
    this.endOfGame = Objects.requireNonNull(endOfGame);
    this.initializeGame = () -> {};
    this.printWinner = () -> {};
  }

  /* Provide setters for customization if you don’t provide
   * constructor for every combination of parameters.
   */
  protected void setGameInitializer(Runnable initializeGame) {
    this.initializeGame = Objects.requireNonNull(initializeGame);
  }

  protected void setWinnerPrinter(Runnable printWinner) {
    this.printWinner = Objects.requireNonNull(printWinner);
  }

  /* A template method calling functional interfaces: */
  public final void playOneGame(int playersCount) {
    this.playersCount = playersCount;
    initializeGame.run();
    int j = 0;
    while (!endOfGame.get()) {
      makePlay.accept(j);
      j = (j + 1) % playersCount;
    }
    printWinner.run();
  }

  // Now we can extend this class in order
  // to implement an actual game.
  // Note that there’s no need to define methods
  // that you don’t want to override. For methods
  // you’d like to override you can either supply
// a functional reference or a lambda:
class Monopoly extends Game {

    public Monopoly() {
        //Initialize parent with functional references
        super(this::makePlay, this::endOfGame);
        // make additional tweaking if you need to.
        // you can even change it in runtime if you want!
        setGameInitializer(this::startGame);
    }

    /*
     * Implementation of concrete method. Note that names
     * can be different now!
     */
    void startGame() {
        // 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
    }

    /* Specific declarations for the Monopoly game. */
    // ...
}

abstract class Game {
    abstract protected void initialize();
    abstract protected void startPlay();
    abstract protected void endPlay();

    /** Template method */
    public final void play() {
        /** Primitive */
        initialize();
        /** Primitive */
        startPlay();
        /** Primitive */
        endPlay();
    }

    class Mario extends Game {
        protected void initialize() {
            echo "Mario Game Initialized! Start playing.", PHP_EOL;
        }

        protected void startPlay() {
            echo "Mario Game Started. Enjoy the game!", PHP_EOL;
        }
    }
}
protected void endPlay()
{
    echo "Mario Game Finished!", PHP_EOL;
}
}
class Tankfight extends Game
{
    protected void initialize()
    {
        echo "Tankfight Game Initialized! Start playing.", PHP_EOL;
    }
    protected void startPlay()
    {
        echo "Tankfight Game Started. Enjoy the game!", PHP_EOL;
    }
    protected void endPlay()
    {
        echo "Tankfight Game Finished!", PHP_EOL;
    }
}
$game = new Tankfight();
$game->play();
$game = new Mario();
$game->play();
case class Song(name: String, lyrics: String, music: String)

trait ConcertPlayer {
    def specialEffectsStart()
    def greetTheAudience()
    def introduceYourSelf()
    def songsIterator(): Iterable[Song]
    def sayGoodbye()
}
final def playConcert() {
    specialEffectsStart()
    greetTheAudience()
    introduceYourSelf()
    songsIterator() foreach { song =>
        println(s"Now we will play the song named ${song.name}")
        println(song.music)
        println(song.lyrics)
    }
    sayGoodbye()
}

class Artist1 extends ConcertPlayer {
    def sayGoodbye() {
        println("See you!")
    }
    def songsIterator(): Iterable[Song] =
    1 to 10 map { index =>
        Song(s"song $index", s"lyrics $index", s"music $index")
    }
implementations

```scala
def introduceYourSelf() {
  println("I'm the Artist1!")
}

def greetTheAudience() {
  println("Hey!")
}

def specialEffectsStart() {
  println("Pyrotechnics...")
}

class Artist2 extends ConcertPlayer {
  def sayGoodbye() {
    println("Bye-Bye")
  }

  def songsIterator(): Iterable[Song] =
    11 to 21 map { index =>
      Song(s"song $index", s"lyrics $index", s"music $index")
    }

  def introduceYourSelf() {
    println("I'm the Artist2!")
  }

  def greetTheAudience() {
    println("Hi!")
  }

  def specialEffectsStart() {
    println("Flames...")
  }
}

object TemplateMethodTest extends App {
  val artist1 = new Artist1
  val artist2 = new Artist2
  artist1.playConcert()
  artist2.playConcert()
}
```

305
24 Visitor

The visitor pattern allows you to easily perform a given operation (function, algorithm, etc...) on each element of a complex structure. The benefit is that you can easily add other operations to perform and you can easily change the structure of the elements. The principle is that the classes that are used to perform the operations and the classes that scroll the structures are different. It also means that, if you know that you will always have only one operation to perform on the structure, this pattern is useless.

Let's take the simple data class:

We want to perform an operation on it. Instead of implementing it inside of the class, we will use a visitor class. A visitor class is a class that performs an action on an object. It will perform an action on a whole structure by being called for each element. However, for now, we only have one object. The visitor will be called by the visit method by the data object, and the object will be called by the accept method by the client:

Then you can define a structure of data object and each object will call the accept() method on its sub-elements. You may think that the main drawback of this pattern is that even if the visitor can process all the elements, it does not process it as a whole, i.e. it
can't connect the different elements together, so the pattern is limited. Not at all. The visitor can keep information about the elements it has already visited (for example to log lines with appropriate indentation). If the visitor needs information from all the elements (for example to compute a percentage based on the sum), it can visit all the elements, store information, and actually process all the information after the last visit using another method. In other words, you can add states to your visitor.

This process is essentially equivalent to getting the collection's iterator\(^1\), and using a \texttt{while(iterator.hasNext())} loop to invoke \texttt{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 \texttt{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 whatever 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 \texttt{foreach} method.

24.1 An example implementation of Visitor Pattern: String

To illustrate the 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 \texttt{char}s that make up the string, and we have a method called \texttt{getCharAt} that takes an \texttt{int} as its only argument, and returns the character at that position in the string, as a \texttt{char}. We also have a method called \texttt{length} that takes no arguments, and returns an \texttt{int} that 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 that will take an instance that implements the \texttt{ICharacterVisitor} interface (that we'll define, below), and calls its \texttt{visit} method for each character in the string. First we need to define what this \texttt{ICharacterVisitor} interface looks like:

```java
public interface ICharacterVisitor {
    public void visit(final char aChar);
}
```

Easy enough. Now, let's get down to our class, which we'll call \texttt{MyString}, and it looks something like this:

```java
public class MyString {
    // … other methods, fields

    // Our main implementation of the visitor pattern
    public void foreach(final ICharacterVisitor aVisitor) {
        // …
    }
}
```

---

\(^1\) Chapter 13 on page 175
Another example implementation of Visitor Pattern: Rock

```java
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. What can we do with this? Well, let's make a class called MyStringPrinter, which prints an instance of MyString to the standard output.

```java
public class MyStringPrinter implements ICharacterVisitor {
    // We have to implement this method because we're implementing the
    ICharacterVisitor
    // interface
    public void visit(final 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(final 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);
    }
}
```

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.

### 24.2 Another example implementation of Visitor Pattern: Rock

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 it is 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 some way for clients to get to 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
Visitor

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 are 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, our visitor interface (assume Rock is already defined somewhere, we don't care what it is or what it does):

```java
public interface IRockVisitor {
    public void visit(final Rock aRock);
}
```

Easy. Now out MyBoxOfRocks

```java
public class MyBoxOfRocks {
    private Rock[] fRocks;
    //... some kind of instantiation code
    public void foreach(final IRockVisitor aVisitor) {
        int length = fRocks.length;
        for (int i = 0; i < length; 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 that 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 value? Infinitesimal, sure, but maybe when you're iterating over 10 million rocks, it makes a difference (maybe!). So, he decides to change the implementation:

```java
public void foreach(final IRockVisitor 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:

```java
public class MyBoxOfRocks {
    // This coder really likes Linked Lists
    private class RockNode {
        Rock iRock;
        RockNode iNext;
    }
    private RockNode[] fRocks;
    //... some kind of instantiation code
    public void foreach(final IRockVisitor aVisitor) {
        int length = fRocks.length;
        for (int i = length - 1; i >= 0; i--) {
            aVisitor.visit(fRocks[i]);
        }
    }
} // End class MyBoxOfRocks
```
Another example implementation of Visitor Pattern: Rock

```java
    RockNode(final Rock aRock, final RockNode aNext) {
        this.iRock = aRock;
        this.iNext = aNext;
    }
    
    private RockNode fFirstRock;
    
    // Our new implementation
    public void foreach(final IRockVisitor 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:

```java
    public void foreach(final IRockVisitor aVisitor) {
        Thread t; // This should speed up the return
        int length = fRocks.length;
        for (int i = length - 1; i >= 0; i--) {
            final Rock current = fRocks[i];
            t = new Thread() {
                public void run() {
                    aVisitor.visit(current);
                }
            }; // End anonymous Thread class
            t.start(); // Run the thread we just created.
        }
    }
```

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, and it will start running, splitting cycles on the processor with other Threads, right away, 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 a long time – 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 "current" in the above code, it's just a bit of technicality using anonymous classes: they can't access non-final local variables. Even though fRocks doesn't fit into this category (it's not local, but an instance variable), 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 it depends on the results of previous calls of visit. Well, the implementation inside the foreach method is encapsulated, so 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, but they will 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.

24.3 Examples

24.4 Cost

This pattern is flexible enough not to block you. At worst, you will need to spend time thinking about how to solve problems, but this pattern will never block you.

24.4.1 Creation

This pattern is expensive to create, if your code already exists.
24.4.2 Maintenance

It is easy to adapt the code with this pattern, even if there are new links between the item visits.

24.4.3 Removal

This pattern can be removed using refactoring operations from your IDE.

24.5 Advises

- Use the *visitor* and *visit* term to indicate the use of the pattern to the other developers.
- If you have and will always have only one visitor, you’d rather implement the *composite* pattern.

24.6 Implementations

The following example is an example in the C# programming language:\(^2\):

```csharp
using System;

namespace VisitorPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            var car = new Car();
            CarElementVisitor doVisitor = new CarElementDoVisitor();
            CarElementVisitor printVisitor = new CarElementPrintVisitor();
            printVisitor.visit(car);
            doVisitor.visit(car);
        }
    }

    public interface CarElementVisitor
    {
        void visit(Body body);
        void visit(Car car);
        void visit(Engine engine);
        void visit(Wheel wheel);
    }

    public interface CarElement
    {
        void accept(CarElementVisitor visitor); // CarElements have to provide accept().
    }

    public class Wheel : CarElement
    {
        public String name { get; set; }
        public void accept(CarElementVisitor visitor)
    }
}
```

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(Body body)
    {
        Console.WriteLine("Visiting body");
    }
    public void visit(Car car)
    {
        Console.WriteLine("Visiting car");
        foreach (var element in car.elements)
        {
            element.accept(this);
        }
        Console.WriteLine("Visited car");
    }
    public void visit(Engine engine)
    {
        Console.WriteLine("Visiting engine");
    }
    public void visit(Wheel wheel)
    {
        Console.WriteLine("Visiting " + wheel.name + " wheel");
    }
}

public class CarElementDoVisitor : CarElementVisitor
{
    public void visit(Body body)
    {
        Console.WriteLine("Moving my body");
    }
    public void visit(Car car)
    {
        // Implementation
    }
}
The following example is in the D programming language³:

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

interface CarElementVisitor {
    void visit(Body bod);
    void visit(Car car);
    void visit(Engine engine);
    void visit(Wheel wheel);
}

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() {
```
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) {
        writeln("Visiting ~ wheel.getName() ~ wheel");
    }
    public void visit(Car car) {
        writeln("\nVisiting car");
        foreach(CarElement element ; car.elements) {
            element.accept(this);
        }
        writeln("Visited car");
    }
    public void visit(Engine engine) {
        writeln("Visiting engine");
    }
    public void visit(Body bod) {
        writeln("Visiting body");
    }
}

class CarElementDoVisitor : CarElementVisitor {
    public void visit(Body bod) {
        writeln("Moving my body");
    }
    public void visit(Car car) {
        writeln("\nStarting my car");
        foreach(CarElement carElement ; car.getElements()) {
            carElement.accept(this);
        }
        writeln("Started car");
    }
    public void visit(Engine engine) {
        writeln("Starting my engine");
    }
    public void visit(Wheel wheel) {
        writeln("Kicking my ~ wheel.name");
    }
}

void main() {
    Car car = new Car;
    CarElementVisitor printVisitor = new CarElementPrintVisitor;
    CarElementVisitor doVisitor = new CarElementDoVisitor;
    printVisitor.visit(car);
    doVisitor.visit(car);
}
The following example is in the Java programming language⁴:

```java
interface CarElementVisitor {
    void visit(Body body);
    void visit(Car car);
    void visit(Engine engine);
    void visit(Wheel wheel);
}

interface CarElement {
    void accept(CarElementVisitor visitor); // CarElements have to provide accept().
}

class Wheel implements CarElement {
    private String name;

    public Wheel(final String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void accept(final CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements CarElement {
    /**
     * Accept the visitor.
     * This method will call the method visit(Engine)
     * and not visit(Wheel) nor visit(Body)
     * because <tt>this</tt> is declared as Engine.
     * That's why we need to define this code in each car element class.
     */
    public void accept(final CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body implements CarElement {
    /**
     * Accept the visitor.
     * This method will call the method visit(Body)
     * and not visit(Wheel) nor visit(Engine)
     * because <tt>this</tt> is declared as Body.
     * That's why we need to define this code in each car element class.
     */
}
```

Visitor

```java
   /*
    * One visitor.
    * You can define as many visitor as you want.
    */
   class CarElementPrintVisitor implements CarElementVisitor {
      public void visit(final Body body) {
         System.out.println("Visiting body");
      }

      public void visit(final Car car) {
         System.out.println("Visiting car");
         for(CarElement element : elements) {
            element.accept(visitor);
         }
         System.out.println("Visited car");
      }

      public void visit(final Engine engine) {
      }

      public void visit(final Wheel wheel) {
         System.out.println("Visiting " + wheel.getName() + " wheel");
      }
   }

   /**
    * Another visitor.
    * Each visitor has one functional purpose.
    */
   class CarElementDoVisitor implements CarElementVisitor {
      public void visit(final Body body) {
         System.out.println("Moving my body");
      }

      public void visit(final Car car) {
         System.out.println("Starting my car");
         for(final CarElement element : elements) {
            element.accept(visitor);
         }
         System.out.println("Started my car");
      }
```
public void visit(final Engine engine) {
    System.out.println("Starting my engine");
}

public void visit(final Wheel wheel) {
    System.out.println("Kicking my " + wheel.getName() + " wheel");
}

1 public class VisitorDemo {
2     public static void main(final String[] arguments) {
3         final CarElement car = new Car();
4         car.accept(new CarElementPrintVisitor());
5         car.accept(new CarElementDoVisitor());
6     }
7 }

(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) ()
    (defmethod do-something ((object wheel) (other-object integer))
        (format t "kicking wheel ~s ~s times~%" object other-object))
    (defmethod do-something ((object wheel) (other-object symbol))
        (format t "kicking wheel ~s symbolically using symbol ~s~%" object other-object)))

(defclass body (auto-part) ()
    (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) ...) )

;; do-something visitations

;; catch all
(defmethod do-something (other-object)
    (format t "don't know how " 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))
Visitor

```
"front-left-wheel")
"front-right-wheel")
"rear-right-wheel")
"rear-right-wheel")

;; 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))

The following example is an example in the Scala programming language:

trait Visitable {
  def accept[T](visit: Visitor[T]): T = visit(this)
}

trait Visitor[T] {
  def apply(visitable: Visitable): T
}

trait Node extends Visitable

trait Operand extends Node

case class IntegerLiteral(value: Long) extends Operand

case class PropertyReference(name: String) extends Operand

trait Operator extends Node

case object Greater extends Operator

case object Less extends Operator

case class ComparisonOperation(left: Operand, op: Operator, right: Operand) extends Node

class NoSqlStringifier extends Visitor[String] {
  def apply(visitable: Visitable): String = visitable match {
    case IntegerLiteral(value) => value.toString
    case PropertyReference(name: String) => name
    case Greater => s"&gt"
    case Less => "&lt"
    case ComparisonOperation(left, operator, right) =>
      s"${left.accept(this)}: { ${operator.accept(this)}: ${right.accept(this)}
    }"
  }
}

class SqlStringifier extends Visitor[String] {
  def apply(visitable: Visitable): String = visitable match {
    case IntegerLiteral(value) => value.toString
    case PropertyReference(name: String) => name
    case Greater => ">"
  }
}

5  https://en.wikibooks.org/wiki/Scala
case Less => "<"
  case ComparisonOperation(left, operator, right) =>
    s"WHERE ${left.accept(this)} ${operator.accept(this)}
  ${right.accept(this)}"
}  
}

object VisitorPatternTest {
  def main(args: Array[String]) {
    val condition: Node = ComparisonOperation(PropertyReference("price"), Greater, IntegerLiteral(12))
    println(s"No sql representation = ${condition.accept(new NoSqlStringifier)}")
    println(s"Sql representation = WHERE price > 12")
  }
}

Output:

No sql representation = price: { &gt: 12 }
Sql representation = WHERE price > 12
25 Development stages
# 26 Contributors

<table>
<thead>
<tr>
<th>Edits</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--1</td>
</tr>
<tr>
<td>1</td>
<td>16@r2</td>
</tr>
<tr>
<td>1</td>
<td>49TL3</td>
</tr>
<tr>
<td>2</td>
<td>A5b1</td>
</tr>
<tr>
<td>2</td>
<td>ACiD2~enwikibooks5</td>
</tr>
<tr>
<td>1</td>
<td>AJCham6</td>
</tr>
<tr>
<td>6</td>
<td>Aardila7</td>
</tr>
<tr>
<td>2</td>
<td>Aaron Rotenberg8</td>
</tr>
<tr>
<td>3</td>
<td>Abdull9</td>
</tr>
<tr>
<td>1</td>
<td>Acuscorp10</td>
</tr>
<tr>
<td>1</td>
<td>Adam Zivner11</td>
</tr>
<tr>
<td>54</td>
<td>Adrignola12</td>
</tr>
<tr>
<td>1</td>
<td>Aesopos13</td>
</tr>
<tr>
<td>1</td>
<td>Aff123a14</td>
</tr>
<tr>
<td>1</td>
<td>Ahoerstemeier15</td>
</tr>
<tr>
<td>2</td>
<td>Alainr34516</td>
</tr>
<tr>
<td>7</td>
<td>Alexbot17</td>
</tr>
<tr>
<td>6</td>
<td>Alexey Vazhnov18</td>
</tr>
<tr>
<td>2</td>
<td>Alexius0819</td>
</tr>
<tr>
<td>1</td>
<td>Allstarecho20</td>
</tr>
</tbody>
</table>

2. [https://en.wikibooks.org/wiki/User:16@r](https://en.wikibooks.org/wiki/User:16@r)
Contributors

1 Almabot
2 Alquantor
1 Alvin-cs
1 Amniarix
2 Andersonvom
2 Andre Engels
37 AndreAdrian
4 Andreas Kaufmann
1 Anna Lincoln
7 Anonymous2420
8 Anthony Appleyard
1 Antonielly
2 Arichnad
1 Army
1 ArthurBot
2 Ary29
2 Ascánder
1 Ashley thomas80
1 Ashlux
2 Audriusa
1 AurelienLourot
1 AutisticCatnip
1 Avicennasis
1 Awesomehuman
1 Ayvengo21

23 https://en.wikibooks.org/w/index.php%3ftitle=User:Alvin-cs&action=edit&redlink=1
27 https://en.wikibooks.org/w/index.php%3ftitle=User:AndreAdrian&action=edit&redlink=1
32 https://en.wikibooks.org/w/index.php%3ftitle=User:Antonielly&action=edit&redlink=1
34 https://en.wikibooks.org/w/index.php%3ftitle=User:Arny&action=edit&redlink=1
41 https://en.wikibooks.org/w/index.php%3ftitle=User:AurelienLourot&action=edit&redlink=1
42 https://en.wikibooks.org/w/index.php%3ftitle=User:AutisticCatnip&action=edit&redlink=1
44 https://en.wikibooks.org/w/index.php%3ftitle=User:Awesomehuman&action=edit&redlink=1
45 https://en.wikibooks.org/w/index.php%3ftitle=User:Ayvengo21&action=edit&redlink=1
3 BOTarate
1 Babaisarkar
1 Bayesianese
1 Beetstra
7 Beland
60 BenFrantzDale
1 Benno
1 Bensaccount
1 Benwing
1 Beryllium
1 BigDunc
1 Blankfrack
2 Bluemoose
5 BluesL
2 Blytkerchan
1 Bobo192
1 Boleslav Bobcik
1 Booler80
2 Borneq
1 BotMultichill
2 BradBeattie
1 Brenont
1 Brian428
1 Brighterorange

Contributors

1 Brooklyn170
2 Btx4071
1 Carlif72
1 CatherineMunro73
3 Ceciop74
7 Cedar10175
1 Chandra.mugraha76
1 Charles Matthews77
1 Cmcginnis78
3 CmdrObot79
2 Cmdrjameson80
2 Coldboyqn~enwikibooks81
3 Colonies Chris82
1 Connelly83
1 CountRazumovsky84
1 Cp11185
1 Craig Pemberton86
2 Crhopkins87
1 Cruccone88
21 Cybercobra89
1 Cyril.wack90
1 Cyrius91
1 Da voli92
2 Damien Cassou93

73 https://en.wikibooks.org/wiki/User:CatherineMunro
92 https://en.wikibooks.org/wiki/User:Da_voli
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Drmies[^119]</td>
</tr>
<tr>
<td>2</td>
<td>Druckles[^120]</td>
</tr>
<tr>
<td>2</td>
<td>DumZiBoT[^121]</td>
</tr>
<tr>
<td>3</td>
<td>Dycedarg[^122]</td>
</tr>
<tr>
<td>4</td>
<td>Dysprosia~enwikibooks[^123]</td>
</tr>
<tr>
<td>7</td>
<td>Eaefremov[^124]</td>
</tr>
<tr>
<td>1</td>
<td>Eastlaw[^125]</td>
</tr>
<tr>
<td>2</td>
<td>Ebswift~enwikibooks[^126]</td>
</tr>
<tr>
<td>1</td>
<td>Ecabiac[^127]</td>
</tr>
<tr>
<td>7</td>
<td>EdC[^128]</td>
</tr>
<tr>
<td>6</td>
<td>Eduard Maltsev[^129]</td>
</tr>
<tr>
<td>1</td>
<td>Edward[^130]</td>
</tr>
<tr>
<td>2</td>
<td>Ehnl[^131]</td>
</tr>
<tr>
<td>4</td>
<td>EmausBot[^132]</td>
</tr>
<tr>
<td>1</td>
<td>Emcmanus[^133]</td>
</tr>
<tr>
<td>1</td>
<td>Emj[^134]</td>
</tr>
<tr>
<td>1</td>
<td>Emufarmers[^135]</td>
</tr>
<tr>
<td>10</td>
<td>Enric Naval[^136]</td>
</tr>
<tr>
<td>1</td>
<td>Epbr123[^137]</td>
</tr>
<tr>
<td>3</td>
<td>Equendil[^138]</td>
</tr>
<tr>
<td>1</td>
<td>Eric-Wester[^139]</td>
</tr>
<tr>
<td>1</td>
<td>Errandir[^140]</td>
</tr>
<tr>
<td>1</td>
<td>Ertwroc[^141]</td>
</tr>
<tr>
<td>4</td>
<td>Eskimbot[^142]</td>
</tr>
<tr>
<td>1</td>
<td>Evil saltine[^143]</td>
</tr>
</tbody>
</table>

Implementations

4  Exe144
1  FatalError145
2  Finlay McWalter146
2  Flammifer~enwikibooks147
1  Flex148
1  Fluffernutter149
2  Forseti150
1  Francesco Terenzani151
2  FrankTobia152
1  Frecklefoot153
9  Fred Bradstadt154
3  Fredrik~enwikibooks155
1  FrescoBot156
1  Frigoris157
1  Frór158
82  Ftiercel159
2  FunnyMan3595~enwikibooks160
4  Furrykef61
1  Fylbecatulous162
1  Fyrael163
1  Garyczek164
1  Genester165
3  Genoskill166
1  Gfxiang167

144  https://en.wikibooks.org/w/index.php%3ftitle=User:Exe&action=edit&redlink=1
147  https://en.wikibooks.org/w/index.php%3ftitle=User:Flammifer~enwikibooks&action=edit&redlink=1
152  https://en.wikibooks.org/w/index.php%3ftitle=User:FrankTobia&action=edit&redlink=1
154  https://en.wikibooks.org/w/index.php%3ftitle=User:Fred_Bradstadt&action=edit&redlink=1
156  https://en.wikibooks.org/w/index.php%3ftitle=User:FrescoBot&action=edit&redlink=1
158  https://en.wikibooks.org/w/index.php%3ftitle=User:Fr%C3%B3r&action=edit&redlink=1
164  https://en.wikibooks.org/w/index.php%3ftitle=User:Garyczek&action=edit&redlink=1
165  https://en.wikibooks.org/w/index.php%3ftitle=User:Genester&action=edit&redlink=1
167  https://en.wikibooks.org/w/index.php%3ftitle=User:Gfxiang&action=edit&redlink=1
Contributors

2 Ggenellina
2 Ghettoblaster
2 GilHamilton
1 Gilliam
1 Gilson111
1 Gluegadget
1 Gmoore19
1 Gnp
1 GoingBatty
3 Gotovikas
2 Grancalavera
5 GregorB
1 Grokus
1 GrouchoBot
1 Guanabot~enwikibooks
1 Gunzapper
6 Gwern
3 Hairy Dude
1 HalfShadow
1 Hangy
19 Hao2lian
2 Hardwigg
1 Hardywang
4 Hariharank12
5 Hephaestos~enwikibooks

Implementations

1. Heron
2. Hervegirod
3. Hipal
4. Hofoen
5. HongxuChen
6. Hooperbloob
7. Hosamaly
8. Hu12
9. IanLewis
10. IanVaughan
11. Igitur
12. ImageObserver
13. Imdonatello
14. Intgr
15. Io41
16. Iridescence
17. Iridescent
18. Itai
19. Ixfd64
20. J.delanoy
21. J04n
22. JBW
23. JHunterJ
24. JLaTondre
25. JMatthews

213 https://en.wikibooks.org/wiki/User:J04n
Contributors

3  JRM$^{218}$
5  JackPotte$^{219}$
1  Jamesontai$^{220}$
4  Jarble$^{221}$
2  Jarsyl~enwikibooks$^{222}$
3  JavierMC$^{223}$
3  Jay~enwikibooks$^{224}$
1  Jdcollins13$^{225}$
2  Jebdm$^{226}$
2  Jeepday$^{227}$
2  Jeeves~enwikibooks$^{228}$
1  Jengelh$^{229}$
1  Jeroen De Dauw$^{230}$
1  Jerryobject$^{231}$
2  Jesus Escaped$^{232}$
1  Jevansen$^{233}$
1  Jldupont~enwikibooks$^{234}$
2  Jlechem$^{235}$
1  Joerg Reiher$^{236}$
4  Jogloran$^{237}$
1  Jokes Free4Me$^{238}$
1  Jonasschneider$^{239}$
1  Jonathan Hall$^{240}$
1  JonathanLivingston~enwikibooks$^{241}$

$^{221}$ https://en.wikibooks.org/wiki/User:Jarble
$^{229}$ https://en.wikibooks.org/wiki/User:Jeroen_De_Dauw
Implementations

18 Jorend~enwikibooks
4 Joriki
2 Joswig
1 Joyous!
1 Jsgoupil
1 Jthiesen~enwikibooks
2 Judgesurreal777
1 Julesd
1 Jussist
1 KJS77~enwikibooks
1 Kallikanzarid
1 Kars~enwikibooks
2 Kate
6 Khalidhassani
1 KhymChanur
1 Kingfishr
1 Kingturtle
3 Kku
1 KlemenKocjancic
1 Komap
1 KurtJansson
1 Kwamikagami
3 LaaknorBot
2 Lababidi

254 https://en.wikibooks.org/wiki/User:Kate
Contributors

1 LanceBarber
266
13 Lathspell~enwikibooks
1 Leaderboard
268
3 Leirith
269
4 Leszek Jańczuk
270
1 Lexxdark
271
2 Liao
272
1 Lightdarkness
273
3 Liko81
274
1 Loadmaster
275
1 Loren.wilton
276
4 Luckas-bot
277
1 Luigi30
278
9 Lukasz.gosik
279
2 Lusum
280
5 Luís Felipe Braga
281
1 M2Ys4U
282
1 M4bwav
283
24 MER-C
284
3 Magioladitis
285
1 Magister Mathematicae
286
10 Mahanga
287
1 Mange01
288
1 Manop
289
10 MarcGarver
290

270 https://en.wikibooks.org/wiki/User:Leszek_Ja%25C5%2584czuk

336
Implementations

1. Marco Krohn
2. Marcusmax
3. Marj Tiefert
4. Marudubshinki
5. MastiBot
2. Materialscientist
2. Maths314
2. Matt Crypto
2. Matěj Grabovský
1. Maxim
1. Md2perpe
1. Mecanismo
1. Merner
5. Merutak
1. Michael Devore
1. Michael Hardy
1. Michael Slone
2. MichalKwiatkowski
2. Mik01aj
1. Mikethegreen
2. Mild Bill Hiccup
2. Minddog
1. Minghong

Contributors

1 Minimac
2 MisterLambda
20 Mital.vora
2 MithrandirMage
1 Mjb
3 Mjchonoles
1 Mjresin
1 Mloskot
1 Modify
1 Modster
9 Mormat
2 MrOllie
6 Mrwojo
1 Murtasa
3 Mxn
3 MystBot
1 Naaram
3 Naasking
2 Nafeezabrar
1 Nakon
3 Narcissus
1 Nate Silva
1 NathanBeach
1 Niceguyedc
1 Nick1915

Implementations

1 Nikai
1 Nishanthan144
1 Nkocharh
1 Nmenezes~enwikibooks
1 Nomad7650
1 Nurg
2 Nzroller~enwikibooks
2 Obersachsebot
1 Oddbodz
1 Oddity-
10 Odie5533
1 Ogmios
1 Oleg Alexandrov
1 Omegatron
7 OrangeDog
1 Orangeroof
1 Ospalh
1 Otterfan~enwikibooks
3 PJonDevelopment
10 Panic2k4
1 Patrickdepinguin
11 Pcb21
1 PeaceNT
3 Pelister

352 https://en.wikibooks.org/wiki/User:Oleg_Alexandrov
Contributors

3 Per Abrahamsen
3 Peterl
3 Pharaoh of the Wizards
1 Phe-bot
3 Philip Baird Shearer
1 Philip Trueman
1 Pi Delport
1 Piano non troppo
1 PierreAbbat
1 Pietrodn
1 Pikiwyn
1 Pinguin.tk
1 Pingveno
1 Platypus222
3 Pmussler
7 Pointillist
18 PokestarFan
7 Ptbotgourou
4 Ptyxs
1 QUBot
1 QuackGuru
1 Quamaretto~enwikibooks
1 R'n'B
1 RJFJR


340
1 RUL3R388
1 Radagast83389
1 Radon210390
1 RainbowOfLight391
1 Ran392
4 Ray Van De Walker393
2 Ray040123394
1 RedCrystal395
26 RedWolf396
1 Redroof397
1 Reinis398
1 RexxS399
4 Reyjose400
1 Rhomboid401
1 Rich Farmbrough402
2 Richard Katz403
5 Rijkbenik404
1 Rimshot405
1 Rishichauhan406
1 Rjnienaber407
1 Rjwilmsi408
1 Rmembrives409
1 Roadrunner~enwikibooks410
1 RobDe68411
1 Robbot412
Contributors

1 RockyMM\(^{413}\)
2 Romanc19s\(^{414}\)
2 Ronabop\(^{415}\)
1 Rory O'Kane\(^{416}\)
1 Rosiestep\(^{417}\)
1 Rossami\(^{418}\)
1 RoyBoy\(^{419}\)
3 Royalguard11\(^{420}\)
1 Rrburke\(^{421}\)
7 Rubinbot\(^{422}\)
1 Rubymage\(^{423}\)
3 Ruud Koot\(^{424}\)
1 Sabre ball\(^{425}\)
1 Sadads\(^{426}\)
43 Sae1962\(^{427}\)
1 Salix alba\(^{428}\)
1 SamePaul\(^{429}\)
1 Satyr\(^{430}\)
1 Sbhukar\(^{431}\)
1 Schulllz\(^{432}\)
1 ScottyBerg\(^{433}\)
2 Scray\(^{434}\)
1 Scullder\(^{435}\)
1 Sebras\(^{436}\)
1 Serious7Sam\(^{437}\)


342
Implementations

2 Shadowjams
3 Shakethehill
1 Shell Kinney
1 Shivendradayal
1 Shyal-b
2 Sietse Snel
4 Sir Nicholas de Mimsy-Porpongton
1 Skapur
2 Skarebo
2 Skittleys
1 Sligocki
1 ShubGhub
12 SmackBot
2 Snaxe920~enwikibooks
5 Sneftel
1 Snooper77
1 Snottywong
1 Solra Bizna
1 SonOfNothing
1 Soubok~enwikibooks
1 Southen
2 SpBot
1 Spookylukey~enwikibooks
2 Spoon!

Contributors

1 Stephenb
1 Strib
1 Strike Eagle
1 Stwalkerster
2 Sun Creator
14 Supadawg~enwikibooks
1 Superborsuk
1 Supersid01
1 Svick
1 Synchronism
1 Synook
1 T638403
1 TKlecka
2 Tabletop
91 TakuyaMurata
3 Tea2min
2 TechBot
1 Tgwizard
1 Thadius856~enwikibooks
1 Thamaraiselvan~enwikibooks
1 The Anome
3 The Thing That Should Not Be
2 The Wild Falcon

484 https://en.wikibooks.org/wiki/User:The_Thing_That_Should_Not_Be

344
Implementations

2 TheMandarin
10 TheParanoidOne
5 Tombomp
1 Tomtheeditor
1 Tony Sidaway
1 Torc2
6 Trashtoy
1 Treutwein
3 Trevor Johns
1 Tubalubalu
3 Tuntable
4 Twsx
3 Umreemala~enwikibooks
1 Uncle C
3 Urhixidur
1 Uzume
4 Vald
1 Valodzka
7 Vermiceli
5 Vinod.pahuja
1 Vipinhari
1 VladimirYefymenko
1 Vltsu
5 VolkovBot
2 Vonkje

Contributors

1 Vvarkey
1 Waggers
2 Wapcaplet~enwikibooks
2 Warren~enwikibooks
1 Wenerme
1 WereSpielChequers
1 West Brom 4ever~enwikibooks
1 Whitepaw
2 Wik~enwikibooks
1 WinerFresh
1 Winstonne
1 Winxa
1 Wlievens
1 Wmwallace
3 Woohookitty
1 Wookietreiber
1 Wwjdcsk
7 XLinkBot
2 Xania
4 XcepticZP
1 Yobot
3 Zarkonnen
1 Zeno Gantner
2 Zerokitsune


346
Implementations

2  Zhen Lin\textsuperscript{536}
2  Zigmar\textsuperscript{537}
1  Zoz\textsuperscript{enwikibooks}\textsuperscript{538}
1  Zyx\textsuperscript{539}

\textsuperscript{536} \url{https://en.wikibooks.org/w/index.php?title=User:Zhen_Lin&action=edit&redlink=1}
\textsuperscript{537} \url{https://en.wikibooks.org/w/index.php?title=User:Zigmar&action=edit&redlink=1}
\textsuperscript{538} \url{https://en.wikibooks.org/w/index.php?title=User:Zoz\textsuperscript{enwikibooks}&action=edit&redlink=1}
\textsuperscript{539} \url{https://en.wikibooks.org/w/index.php?title=User:Zyx&action=edit&redlink=1}
List of Figures

- cc-by-sa-4.0: Creative Commons Attribution ShareAlike 4.0 License. https://creativecommons.org/licenses/by-sa/4.0/deed.en
- 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-1.0: Creative Commons Attribution 1.0 License. https://creativecommons.org/licenses/by/1.0/deed.en
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- 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
- cc-by-4.0: Creative Commons Attribution 4.0 License. https://creativecommons.org/licenses/by/4.0/deed.de
- 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.

- CFR: Copyright free use.

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses\textsuperscript{540}. 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 web browser.
## List of Figures

<table>
<thead>
<tr>
<th></th>
<th>Author(s)</th>
<th>CC License</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Doktor Mandrake, Doktor Mandrake</td>
<td>CC-BY-SA-3.0</td>
</tr>
<tr>
<td>2</td>
<td>Amnon Eden</td>
<td>PD</td>
</tr>
<tr>
<td>3</td>
<td>Giacomo Ritucci, Giacomo Ritucci</td>
<td>CC-BY-SA-3.0</td>
</tr>
<tr>
<td>4</td>
<td>Spischot, Spischot</td>
<td>PD</td>
</tr>
<tr>
<td>5</td>
<td>Econt, Econt</td>
<td>PD</td>
</tr>
<tr>
<td>6</td>
<td>Econt, Econt</td>
<td>PD</td>
</tr>
<tr>
<td>7</td>
<td>Spischot, Spischot</td>
<td>PD</td>
</tr>
<tr>
<td>8</td>
<td>Spischot, Spischot</td>
<td>PD</td>
</tr>
<tr>
<td>9</td>
<td>Spischot, Spischot</td>
<td>PD</td>
</tr>
<tr>
<td>10</td>
<td>Spischot, Spischot</td>
<td>PD</td>
</tr>
<tr>
<td>11</td>
<td>Spischot, Spischot</td>
<td>PD</td>
</tr>
<tr>
<td>12</td>
<td>Spischot, Spischot</td>
<td>PD</td>
</tr>
<tr>
<td>13</td>
<td>Trashtoy, Trashtoy</td>
<td>PD</td>
</tr>
<tr>
<td>14</td>
<td>Sae1962, Sae1962</td>
<td>CC-BY-SA-4.0</td>
</tr>
<tr>
<td>15</td>
<td>vniklos document foundation</td>
<td>PD</td>
</tr>
</tbody>
</table>

---

541 http://commons.wikimedia.org/wiki/User:DoktorMandrake
542 http://commons.wikimedia.org/wiki/User_talk:DoktorMandrake
543 https://wiki/User:DoktorMandrake
544 https://wiki/User_talk:DoktorMandrake
546 http://commons.wikimedia.org/wiki/User:Giacomo_Ritucci
547 https://wiki/User:Giacomo_Ritucci
548 http://commons.wikimedia.org/wiki/User:Spischot
549 https://wiki/User:Spischot
550 http://commons.wikimedia.org/wiki/User:Econt
551 https://wiki/User:Econt
552 http://commons.wikimedia.org/wiki/User:Econt
553 https://wiki/User:Econt
554 http://commons.wikimedia.org/wiki/User:Spischot
555 https://wiki/User:Spischot
556 http://commons.wikimedia.org/wiki/User:Spischot
557 https://wiki/User:Spischot
558 http://commons.wikimedia.org/wiki/User:Spischot
559 https://wiki/User:Spischot
560 http://commons.wikimedia.org/wiki/User:Spischot
561 https://wiki/User:Spischot
562 http://commons.wikimedia.org/wiki/User:Spischot
563 https://wiki/User:Spischot
564 http://commons.wikimedia.org/wiki/User:Spischot
565 https://wiki/User:Spischot
566 http://commons.wikimedia.org/wiki/User:Trashtoy
567 https://wiki/User:Trashtoy
569 https://wiki/User:Sae1962
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Composite_UML_class_diagram.svg Trashtoy derivative work: « Aaron Rotenberg « Talk «</td>
<td>PD</td>
</tr>
<tr>
<td>17</td>
<td>vmiklos document foundation</td>
<td>PD</td>
</tr>
<tr>
<td>18</td>
<td>Ftiercel Ftiercel</td>
<td>CC-BY-3.0</td>
</tr>
<tr>
<td>19</td>
<td>Kostyantyn Yuriyovich Kolesnichenko Kostyantyn Yuriyovich Kolesnichenko</td>
<td>PD</td>
</tr>
<tr>
<td>20</td>
<td>WikiSolved WikiSolved</td>
<td>PD</td>
</tr>
<tr>
<td>23</td>
<td>User:Giacomo Ritucci User:Giacomo Ritucci</td>
<td>CC-BY-SA-3.0</td>
</tr>
<tr>
<td>24</td>
<td>Bocsika Bocsika</td>
<td>PD</td>
</tr>
<tr>
<td>25</td>
<td>Jkost Jkost Original uploader was Jkost at en.wikibooks</td>
<td>PD</td>
</tr>
<tr>
<td>28</td>
<td>en:User:JKost en:User:JKost Original uploader was Jkost at en.wikibooks</td>
<td>PD</td>
</tr>
</tbody>
</table>

570 http://commons.wikimedia.org/wiki/File:Composite_UML_class_diagram.svg
571 http://commons.wikimedia.org/wiki/User:Trashtoy
572 http://commons.wikimedia.org/wiki/User:Aaron_Rotenberg
573 http://commons.wikimedia.org/wiki/User_talk:Aaron_Rotenberg
574 https://wiki/File:Composite_UML_class_diagram.svg
575 https://wiki/User:Trashtoy
576 https://wiki/User:Aaron_Rotenberg
577 https://wiki/User_talk:Aaron_Rotenberg
580 http://commons.wikimedia.org/wiki/User:Mormat
581 https://wiki/User:Mormat
586 http://commons.wikimedia.org/wiki/User:Giacomo_Ritucci
587 https://wiki/User:Giacomo_Ritucci
592 https://en.wikibooks.org
597 https://en.wikibooks.org
<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>No machine-readable author provided. MarianSigler(^{600}) assumed (based on copyright claims). No machine-readable author provided. MarianSigler(^{600}) assumed (based on copyright claims).</td>
</tr>
<tr>
<td>31</td>
<td>No machine-readable author provided. MarianSigler(^{600}) assumed (based on copyright claims). No machine-readable author provided. MarianSigler(^{600}) assumed (based on copyright claims).</td>
</tr>
<tr>
<td>32</td>
<td>SVG: Beao(^{603}), Raster: Tarikash(^{603}). SVG: Beao(^{603}), Raster: Tarikash(^{603}).</td>
</tr>
<tr>
<td>33</td>
<td>Spischot(^{609}), Spischot(^{610}).</td>
</tr>
</tbody>
</table>

600 [https://wiki/User:MarianSigler](https://wiki/User:MarianSigler)
The "Corresponding Source" for a work in object code form means all the source code for the work, including components or portions of the work which are not normally distributed under the terms of a patent license, but which are included either at your option or as a consequence of having acquired one or more copyrighted materials under this License, as specified in section 3. This license automatically applies if the corresponding object code is made available under the terms of section 4. This license supplements, does not replace, any other system of attribution or notice to the original author as provided for under section 3 of the copyleft. The Corresponding Source for a work in object code form is the same as the source code for the work, since the corresponding object code is in the public domain. The "source code" for a work means the preferred form of the work for making modifications. The "source code" for an object code work is the work without any material added under section 7. This definition of "source code" does not include any material added under section 7 if you choose not to include it in the Corresponding Source. For an executable work, the "source code" includes both the normal form of the code that you receive as well as any changes you may have added to this work. For an object code work, the "source code" means all the source code for the work, whether or not it is distributed under the terms of this License. In the following three paragraphs, a "Corresponding Source" means all the source code for a work, or portions thereof, to which you made changes and which were added to, or removed from, the version as a whole, as well as all the source code for any of such added or removed parts, if you would have to make changes in the entire version as a material unit to make them usable together. The "source code" for a work means all the source code for all the parts of the work, whether or not distributed under this License. The "source code" for a work means the work as it is in the public domain, as well as any changes you may have added under section 7. A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or is specified and documented in a manner that provides significant information about its meaning and purpose. A "covered work" means either the unmodified Program or a work that has been modified under section 4 or by somebody who has received a copy of the Corresponding Source under a corresponding offer from you, if the work was supplied under the terms of this License by the copyright holder in accord with subsection 1 of section 3. A "Work of the Author" means a "covered work" on which you work, and grant a patent license to some of the parties receiving the "Work of the Author". You may not impose any additional restrictions on the exercise of the permissions in this section if you distribute the Corresponding Source as permitted in section 7. A "Work of the Author" means a "covered work" on which you work, and grant a patent license to some of the parties receiving the "Work of the Author". A "Work of the Author" means a "covered work" on which you work, and grant a patent license to some of the parties receiving the "Work of the Author". You may not impose any additional restrictions on the exercise of the permissions in this section if you distribute the Corresponding Source as permitted in section 7. A "Work of the Author" means a "covered work" on which you work, and grant a patent license to some of the parties receiving the "Work of the Author".

The "source code" for a work means all the source code for all the parts of the work, whether or not distributed under this License. A 'patent license' is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted. A "patent license" is any express permission or covenant to use, propagate, and redistribute the corresponding source code. You may not be able to use the Source Code if your rights have been exhausted.
27.2 GNU Free Documentation License

Version 1.3, 1 November 2008


Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This License is designed for free software and free documentation in the sense of the Free Software Foundation’s Gu-

nu General Public License (GPL) and the GNU Lesser General Public License (LGPL).

The GNU GPL and GPL-like licenses are designed to ensure that writers of primary documents have liberal permis-
sions to publish their documents in any way they choose, including having them printed for sale, so that your re-per-
spectives, if you distribute verbatim copies of the whole Document, you must also follow the rules therein, includ-
ing these. If you use the whole Document as part of a larger text, you may distribute this larger text under the same terms,
provided that you identify the larger text as such.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the pro-
gram are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eneral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.

This License is designed to make portions of the larger whole free by making the source code available to the pub-
lic. The GPL and the LGPL require that the source code be made available if certain executable forms of the program are made available. But in fact all常用的Free Software License permits the release of source when it is re-
duced to a form that enables anyone to make the whole program available for free.

When you distribute household and small business uses. This License is most similar to the GNU Affero Gen-

eral Public License but with one important difference: it permits the creation of Opaque formats. The GNU GPL

and the GNU Lesser General Public License both permit the creation of new Opaque formats for non-pro-

fit purposes, but not for commercial purposes. This License permits Opaque formats to be used for any purpose.

This License is designed so that a work covered by it may be a selection, a secondary version, or multiple identical

versions of the whole original work, selected or combined by you and distributed instead of the original work.
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 must cause a covered work under section 1 or 2, and all the source code in that work, to be licensed under the terms of version 3 of the GNU General Public License as published by the Free Software Foundation.

If you modify a copy of the Library, and if you license that copy of the Library by section 3 above (or modify terms of Creative Commons License, Public Domain Dedication, or other License), create a “Linked Version” of the modified version selected by you, use the following naming scheme: “<name of old version> modified by <name of new owner>.”

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 receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.

You may convey a Combined Work under terms of your choice, that takes together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and receive engineering for debugging such modifications, if you do so at your own expense.