FREE eBook

LEARNING

Design patterns

Free unaffiliated eBook created from
Stack Overflow contributors.

#design-patterns

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: design-patterns

It is an unofficial and free Design patterns ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Design patterns.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Design patterns

Remarks

This section provides an overview of what design-patterns is, and why a developer might want to use it. Examples may provide a graphical representation of the pattern, a scenario consisting of a problem given a context in which a pattern can be used and mention possible trade offs.

It should also mention any large subjects within design-patterns, and link out to the related topics. Since the Documentation for design-patterns is new, you may need to create initial versions of those related topics.

Examples

Introduction

According to Wikipedia:

> [A] *software design pattern* is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.
>
> (Retrieved: 2016-10-13)

There are many recognized software design patterns, and new ones are proposed on a regular basis. Other topics cover many of the most common patterns, and the Wikipedia article provides a more extensive list.

Similarly, there are different ways to classify design patterns, but the original classification is:

- **Creational patterns**: Factory, Builder, Singleton, etc.
- **Structural patterns**: Adapter, Composite, Proxy, etc.
- **Behavioral patterns**: Iterator, Strategy, Visitor, etc.
- **Concurrency patterns**: ActiveObject, Monitor, etc.

The idea of design patterns has been extended to *domain-specific design patterns* for domains such as user interface design, data visualization, secure design, web design and business model design.

Finally, there is a related concept called a *software architecture pattern* which is described as being the analogue for design patterns applied to software architectures.

Read Getting started with Design patterns online: https://riptutorial.com/design-patterns/topic/1012/getting-started-with-design-patterns

Chapter 2: Adapter

Examples

Adapter Pattern (PHP)

A real world example using a scientific experiment where certain routines are performed on different types of tissue. The class contains two functions by default to get the tissue or routine separately. In a later version we have then adapted it using a new class to add a function that gets both. This means we have not edited the original code and therefore do not run any risk of breaking our existing class (and no retesting).

```
class Experiment {
    private $routine;
    private $tissue;
    function __construct($routine_in, $tissue_in) {
        $this->routine = $routine_in;
        $this->tissue  = $tissue_in;
    }
    function getRoutine() {
        return $this->routine;
    }
    function getTissue() {
        return $this->tissue;
    }
}

class ExperimentAdapter {
    private $experiment;
    function __construct(Experiment $experiment_in) {
        $this->experiment = $experiment_in;
    }
    function getRoutineAndTissue() {
        return $this->experiment->getTissue().' (. $this->experiment->getRoutine().')';
    }
}
```

Adapter (Java)

Lets assume that in your current codebase, there exists `MyLogger` interface like so:

```
interface MyLogger {
    void logMessage(String message);
    void logException(Throwable exception);
}
```

Lets say that you've created a few concrete implementations of these, such as `MyFileLogger` and `MyConsoleLogger`.

You have decided that you want to use a framework for controlling your application's Bluetooth connectivity. This framework contains a `BluetoothManager` with the following constructor:

```
class BluetoothManager {
    private FrameworkLogger logger;

    public BluetoothManager(FrameworkLogger logger) {
        this.logger = logger;
    }
}
```

The `BluetoothManager` also accepts a logger, which is great! However it expects a logger of which the interface was defined by the framework and they have used method overloading instead of naming their functions differently:

```
interface FrameworkLogger {
    void log(String message);
    void log(Throwable exception);
}
```

You already have a bunch of `MyLogger` implementations that you would like to reuse, but they do not fit the interface of the `FrameworkLogger`. This is where the adapter design-pattern comes in:

```
class FrameworkLoggerAdapter implements FrameworkLogger {
    private MyLogger logger;

    public FrameworkLoggerAdapter(MyLogger logger) {
        this.logger = logger;
    }

    @Override
    public void log(String message) {
        this.logger.logMessage(message);
    }

    @Override
    public void log(Throwable exception) {
        this.logger.logException(exception);
    }
}
```

By defining an adapter class that implements the `FrameworkLogger` interface and accepts a `MyLogger` implementation the functionality can be mapped between the different interfaces. Now it is possible to use the `BluetoothManager` with all of the `MyLogger` implementations like so:

```
FrameworkLogger fileLogger = new FrameworkLoggerAdapter(new MyFileLogger());
BluetoothManager manager = new BluetoothManager(fileLogger);

FrameworkLogger consoleLogger = new FrameworkLoggerAdapter(new MyConsoleLogger());
BluetoothManager manager2 = new BluetoothManager(consoleLogger);
```

Java Example

A great existing example of the Adapter pattern can be found in the SWT MouseListener and MouseAdapter classes.

The MouseListener interface looks as follows:

```
public interface MouseListener extends SWTEventListener {
    public void mouseDoubleClick(MouseEvent e);
    public void mouseDown(MouseEvent e);
    public void mouseUp(MouseEvent e);
}
```

Now imagine a scenario where you are building a UI and adding these listeners, but most of the time you don't care about anything other than when something is single clicked (mouseUp). You wouldn't want to constantly be creating empty implementations:

```
obj.addMouseListener(new MouseListener() {

    @Override
    public void mouseDoubleClick(MouseEvent e) {
    }

    @Override
    public void mouseDown(MouseEvent e) {
    }

    @Override
    public void mouseUp(MouseEvent e) {
        // Do the things
    }

});
```

Instead, we can use MouseAdapter:

```
public abstract class MouseAdapter implements MouseListener {
    public void mouseDoubleClick(MouseEvent e) { }
    public void mouseDown(MouseEvent e) { }
    public void mouseUp(MouseEvent e) { }
}
```

By providing empty, default implementations, we are free to override only those methods which we care about from the adapter. Following from the above example:

```
obj.addMouseListener(new MouseAdapter() {

    @Override
    public void mouseUp(MouseEvent e) {
        // Do the things
    }

});
```

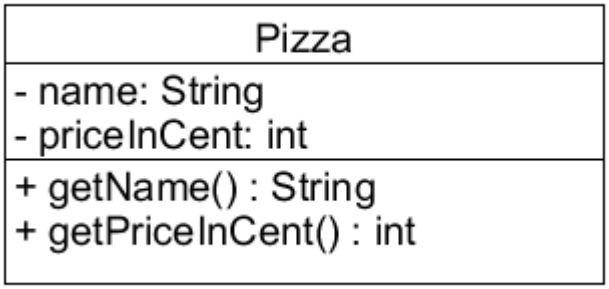
Adapter (UML & example situation)

To make the use of the adapter pattern and the kind of situation when it may be applied more imaginable, a small, simple and very concrete example is given here. There will be no code in here, just UML and a description of the example situation and its problem. Admittedly, the UML

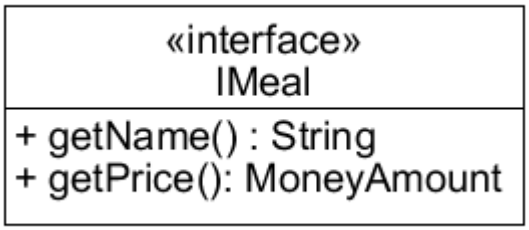
content is written like Java. (Well, the hint text said "Good examples are mostly code", I think design patterns are abstract enough to be introduced in a different way, too.)

In general, the adapter pattern is an adequate solution for a situation when you have incompatible interfaces and none of them can be directly rewritten.

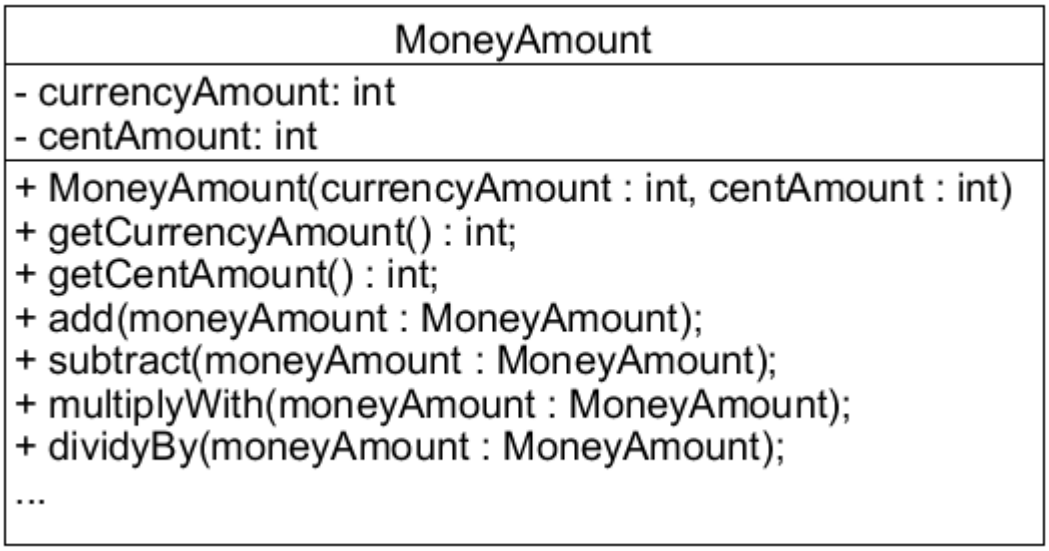
Imagine you're running a nice little pizza delivery service. Customers can order online on your website and you have small system using a class `Pizza` to represent your pizzas and calculate bills, tax reports and more. The price of your pizzas is given as a single integer representing the price in cent (of the currency of your choice).



Your delivery service is working out great, but at some point you cannot handle the growing number of customers on your own anymore but you still want to expand. You decide to add your pizzas to the menu of a big online meta delivery service. They offer a lot of different meals — not only pizzas — so their system makes more use of abstraction and has an Interface `IMeal` representing meals coming along with a class `MoneyAmount` representing money.



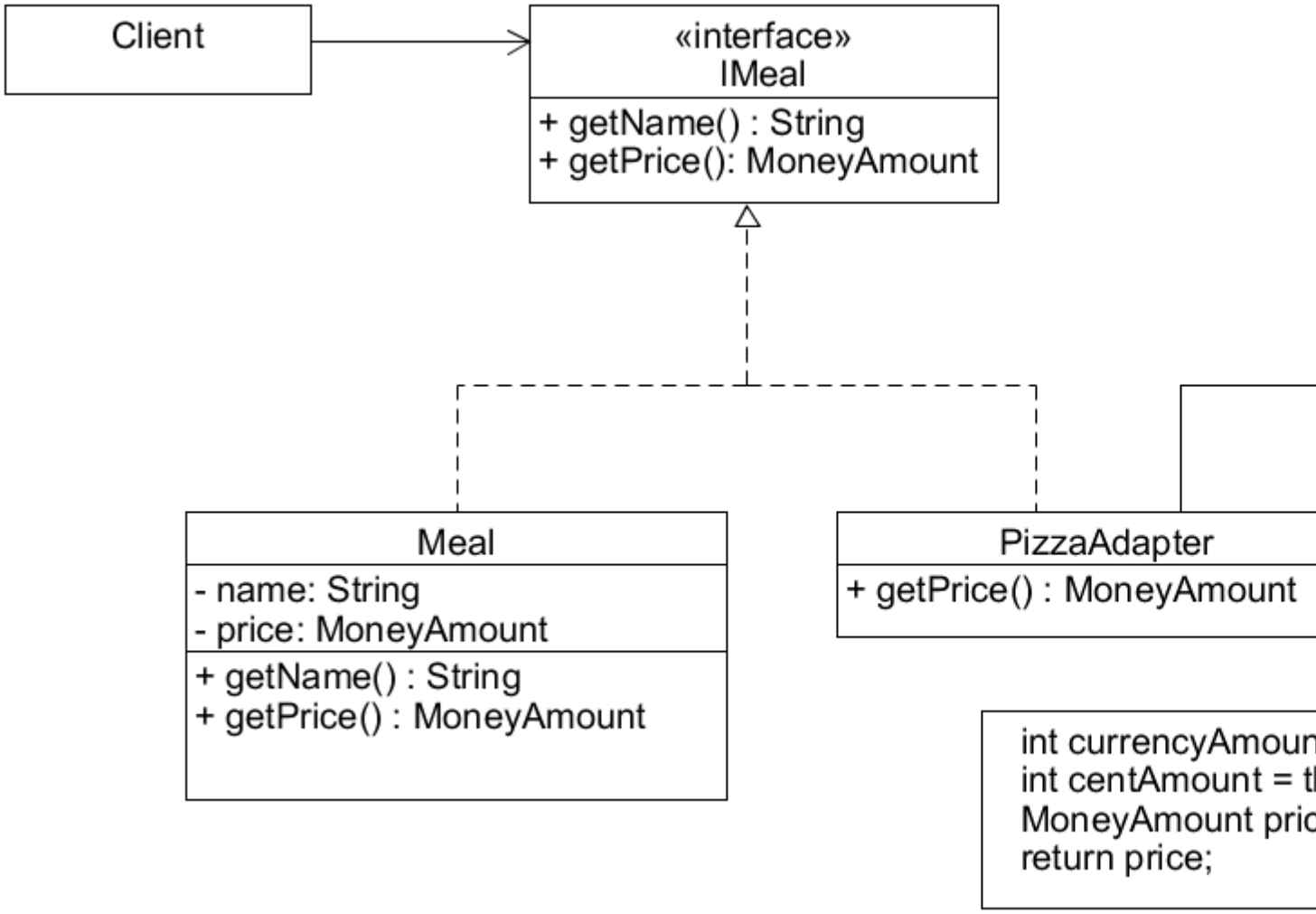
`MoneyAmount` consists of two integers as input, one for the amount (or some random currency) before the comma, and one for the cent amount from 0 to 99 after the comma;



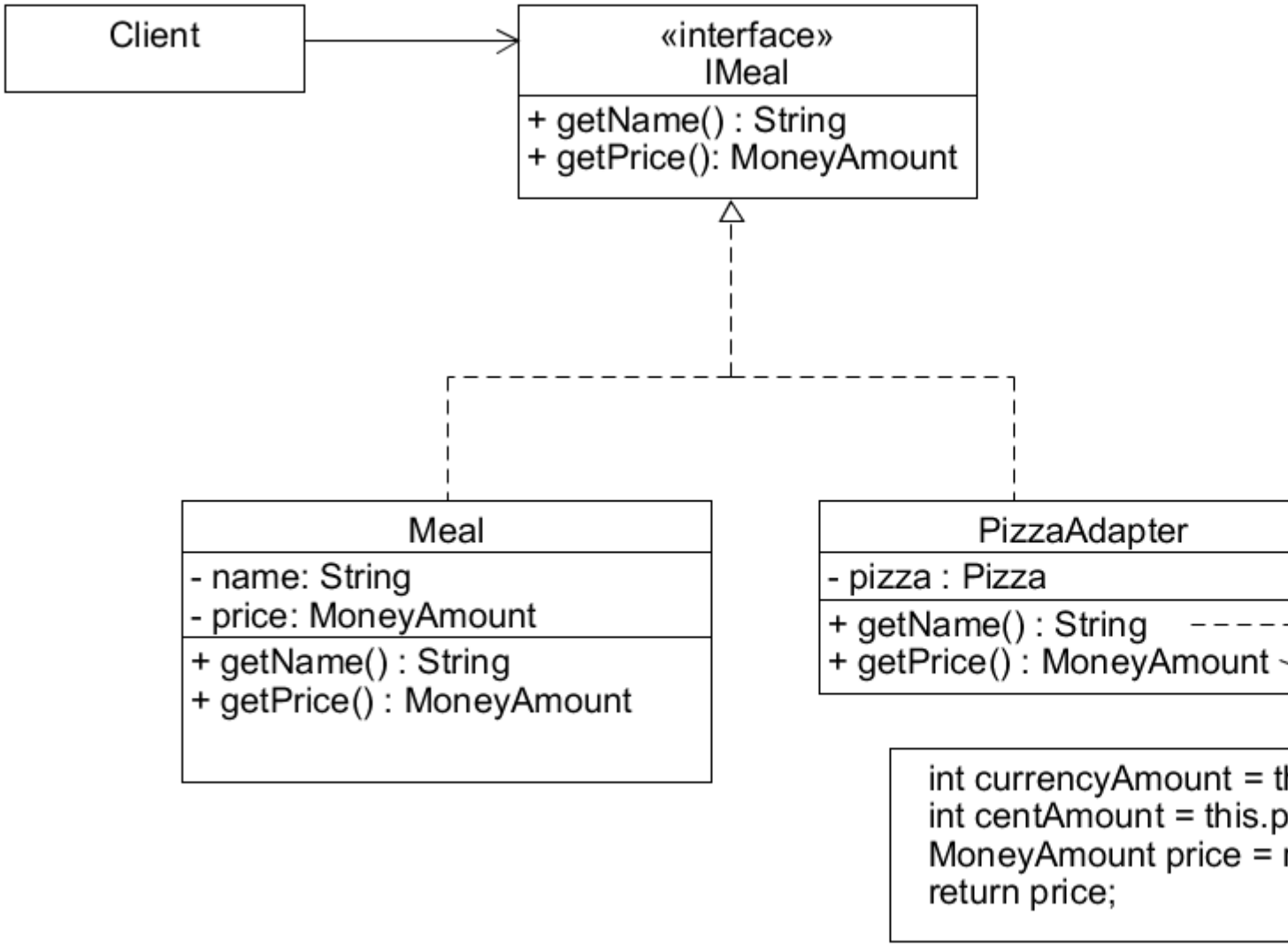
Due to the fact that the price of your `Pizza` is a single integer representing the total price as an amount of cent (> 99), it is not compatible with `IMeal`. This is the point where the adapter pattern comes into play: In case it would take too much effort to change your own system or create a new one and you have to implement an incompatible interface, you may want to apply the adapter patttern.

There are two ways of applying the pattern: class adapter and object adapter.

Both have in common that an adapter (`PizzaAdapter`) works as some kind of translator between the new interface and the adaptee (`Pizza` in this example). The adapter implements the new interface (`IMeal`) and then either inherits from `Pizza` and converts its own price from one integer to two (class adapter)



or has an object of type `Pizza` as an attribute and converts the values of that (object adapter).



By applying the adapter pattern, you will kind of "translate" between incompatible interfaces.

Read Adapter online: https://riptutorial.com/design-patterns/topic/4580/adapter

Chapter 3: blackboard

Examples

C# Sample

Blackboard.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;


namespace Blackboard
{
    public class BlackBoard
    {
        public List<KnowledgeWorker> knowledgeWorkers;
        protected Dictionary<string, ControlData> data;
        public Control control;


        public BlackBoard()
        {
            this.knowledgeWorkers = new List<KnowledgeWorker>();
            this.control = new Control(this);
            this.data = new Dictionary<string, ControlData>();
        }

        public void addKnowledgeWorker(KnowledgeWorker newKnowledgeWorker)
        {
            newKnowledgeWorker.blackboard = this;
            this.knowledgeWorkers.Add(newKnowledgeWorker);
        }

        public Dictionary<string, ControlData> inspect()
        {
            return (Dictionary<string, ControlData>) this.data.ToDictionary(k => k.Key, k =>
(ControlData) k.Value.Clone());
        }
        public void update(KeyValuePair<string, ControlData> blackboardEntry)
        {
            if (this.data.ContainsKey(blackboardEntry.Key))
            {
                this.data[blackboardEntry.Key] = blackboardEntry.Value;
            }
            else
                throw new InvalidOperationException(blackboardEntry.Key + " Not Found!");
        }

        public void update(string key, ControlData data)
        {
            if (this.data.ContainsKey(key))
            {
```

```
                this.data[key] = data;
            }
            else
            {
                this.data.Add(key, data);
            }
        }

        public void print()
        {
            System.Console.WriteLine("Blackboard state");
            foreach (KeyValuePair<string, ControlData> cdata in this.data)
            {
                Console.WriteLine(string.Format("data:{0}", cdata.Key));
                Console.WriteLine(string.Format("\tProblem:{0}", cdata.Value.problem));
                if(cdata.Value.input!=null)
                    Console.WriteLine(string.Format("\tInput:{0}",
string.Join(",",cdata.Value.input)));
                if(cdata.Value.output!=null)
                    Console.WriteLine(string.Format("\tOutput:{0}",
string.Join(",",cdata.Value.output)));
            }
        }

    }
}
```

Control.cs

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


namespace Blackboard
{
    public class Control
    {
        BlackBoard blackBoard = null;

        public Control(BlackBoard blackBoard)
        {
            this.blackBoard = blackBoard;
        }

        public void loop()
        {
            System.Console.WriteLine("Starting loop");
            if (blackBoard == null)
                throw new InvalidOperationException("blackboard is null");
            this.nextSource();
            System.Console.WriteLine("Loop ended");

        }

        /// <summary>
        /// Selects the next source of knowledge (knowledgeworker by inspecting the
blackgoard)
        /// </summary>
        void nextSource()
```

```
        {
            // observers the blackboard
            foreach (KeyValuePair<string, ControlData> value in this.blackBoard.inspect())
            {
                if (value.Value.problem == "PrimeNumbers")
                {
                    foreach (KnowledgeWorker worker in this.blackBoard.knowledgeWorkers)
                    {
                        if (worker.getName() == "PrimeFinder")
                        {
                            Console.WriteLine("Knowledge Worker Found");
                            worker.executeCondition();
                            worker.executeAction();
                            worker.updateBlackboard();
                        }
                    }
                }
            }
        }
    }
}
```

ControlData.cs

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


namespace Blackboard
{
    public class ControlData:ICloneable
    {
        public string problem;
        public object[] input;
        public object[] output;
        public string updateby;
        public DateTime updated;


        public ControlData()
        {
            this.problem = null;
            this.input = this.output = null;
        }

        public ControlData(string problem, object[] input)
        {
            this.problem = problem;
            this.input = input;
            this.updated = DateTime.Now;
        }

        public object getResult()
        {
            return this.output;
        }

        public object Clone()
        {
```

```
            ControlData clone;
            clone = new ControlData(this.problem, this.input);
            clone.updated = this.updated;
            clone.updateby = this.updateby;
            clone.output = this.output;
            return clone;
        }
    }
}
```

KnowledgeWorker.cs

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

namespace Blackboard {
    /// <summary>
    /// each knowledgeworker is resposible for knowing the conditions under which it can
contribute to a solution.
    /// </summary>
    abstract public  class KnowledgeWorker
    {
        protected Boolean canContribute;
        protected string Name;
        public BlackBoard blackboard = null;
        protected List<KeyValuePair<string, ControlData>> keys;
        public KnowledgeWorker(BlackBoard blackboard, String Name)
        {
            this.blackboard = blackboard;
            this.Name = Name;
        }

        public KnowledgeWorker(String Name)
        {
            this.Name = Name;
        }

        public string getName()
        {
            return this.Name;
        }

        abstract public void executeAction();

        abstract public void executeCondition();

        abstract public void updateBlackboard();

    } }
```

Read blackboard online: https://riptutorial.com/design-patterns/topic/6519/blackboard

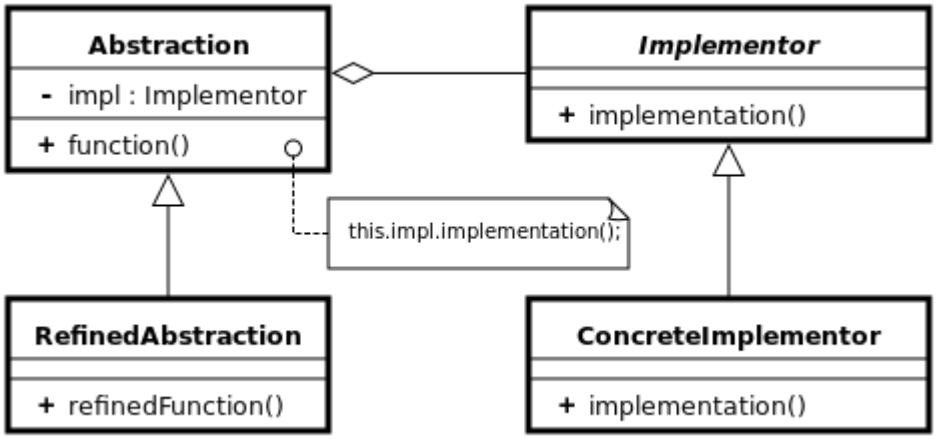
Chapter 4: Bridge Pattern

Examples

Bridge pattern implementation in java

Bridge pattern decouples abstraction from implementation so that both can vary independently. It has been achieved with composition rather than inheritance.

Bridge UML diagram from wikipedia:



You have four components in this pattern.

`Abstraction`: It defines an interface

`RefinedAbstraction`: It implements abstraction:

`Implementor`: It defines an interface for implementation

`ConcreteImplementor`: It implements Implementor interface.

The crux of Bridge pattern : Two orthogonal class hierarchies using composition (and no inheritance). The Abstraction hierarchy and Implementation hierarchy can vary independently. Implementation never refers Abstraction. Abstraction contains Implementation interface as a member (through composition). This composition reduces one more level of inheritance hierarchy.

Real word Use case:

Enable different vehicles to have both versions of manual and auto gear system.

Example code:

```
/* Implementor interface*/
interface Gear{
    void handleGear();
```

```
}

/* Concrete Implementor - 1 */
class ManualGear implements Gear{
    public void handleGear(){
        System.out.println("Manual gear");
    }
}
/* Concrete Implementor - 2 */
class AutoGear implements Gear{
    public void handleGear(){
        System.out.println("Auto gear");
    }
}
/* Abstraction (abstract class) */
abstract class Vehicle {
    Gear gear;
    public Vehicle(Gear gear){
        this.gear = gear;
    }
    abstract void addGear();
}
/* RefinedAbstraction - 1*/
class Car extends Vehicle{
    public Car(Gear gear){
        super(gear);
        // initialize various other Car components to make the car
    }
    public void addGear(){
        System.out.print("Car handles ");
        gear.handleGear();
    }
}
/* RefinedAbstraction - 2 */
class Truck extends Vehicle{
    public Truck(Gear gear){
        super(gear);
        // initialize various other Truck components to make the car
    }
    public void addGear(){
        System.out.print("Truck handles " );
        gear.handleGear();
    }
}
/* Client program */
public class BridgeDemo {
    public static void main(String args[]){
        Gear gear = new ManualGear();
        Vehicle vehicle = new Car(gear);
        vehicle.addGear();

        gear = new AutoGear();
        vehicle = new Car(gear);
        vehicle.addGear();

        gear = new ManualGear();
        vehicle = new Truck(gear);
        vehicle.addGear();

        gear = new AutoGear();
        vehicle = new Truck(gear);
```

```
        vehicle.addGear();
    }
}
```

output:

```
Car handles Manual gear
Car handles Auto gear
Truck handles Manual gear
Truck handles Auto gear
```

Explanation:

1. `Vehicle` is an abstraction.
2. `Car` and `Truck` are two concrete implementations of `Vehicle`.
3. `Vehicle` defines an abstract method : `addGear()`.
4. `Gear` is implementor interface
5. `ManualGear` and `AutoGear` are two implementations of `Gear`
6. `Vehicle` contains `implementor` interface rather than implementing the interface. `Compositon` of implementor interface is crux of this pattern : *It allows abstraction and implementation to vary independently.*
7. `Car` and `Truck` define implementation (redefined abstraction) for abstraction : `addGear()` : It contains `Gear` - Either `Manual` or `Auto`

Use case(s) for Bridge pattern:

1. **Abstraction** and **Implementation** can change independent each other and they are not bound at compile time
2. Map orthogonal hierarchies - One for *Abstraction* and one for *Implementation*.

Read Bridge Pattern online: https://riptutorial.com/design-patterns/topic/4011/bridge-pattern

Chapter 5: Builder Pattern

Remarks

Separates the construction of a complex object from its representation so that the same construction process can create different representations.

- Separate the logic from representation.
- Reuse logic to work with different set of data.

Examples

Builder Pattern / C# / Fluent Interrface

```
public class Email
{
    public string To { get; set; }
    public string From { get; set; }
    public string Subject { get; set; }
    public string Body { get; set; }
}

public class EmailBuilder
{
    private readonly Email _email;

    public EmailBuilder()
    {
        _email = new Email();
    }

    public EmailBuilder To(string address)
    {
        _email.To = address;
        return this;
    }

    public EmailBuilder From(string address)
    {
        _email.From = address;
        return this;
    }

    public EmailBuilder Subject(string title)
    {
        _email.Subject = title;
        return this;
    }

    public EmailBuilder Body(string content)
    {
        _email.Body = content;
        return this;
    }
```

```
    public Email Build()
    {
        return _email;
    }
}
```

Usage example:

```
var emailBuilder = new EmailBuilder();
var email = emailBuilder
    .To("email1@email.com")
    .From("email2@email.com")
    .Subject("Email subject")
    .Body("Email content")
    .Build();
```

Builder Pattern / Java Implementation

The Builder pattern allows you to create an instance of a class with many optional variables in an easy to read way.

Consider the following code:

```
public class Computer {

    public GraphicsCard graphicsCard;
    public Monitor[] monitors;
    public Processor processor;
    public Memory[] ram;
    //more class variables here...

    Computer(GraphicsCard g, Monitor[] m, Processer p, Memory ram) {
        //code omitted for brevity...
    }

    //class methods omitted...

}
```

This is all well and good if all of the parameters are necessary. What if there are a lot more variables and/or some of them are optional? You don't want to create a large number of constructors with each possible combination of required and optional parameters because it becomes difficult to maintain and for developers to understand. You also may not want to have a long list of parameters in which many may need to be entered as null by the user.

The Builder pattern creates an inner class called Builder that is used to instantiate only the desired optional variables. This is done through methods for each optional variable which take the variable type as a parameter and return a Builder object so that the methods can be chained with each other. Any required variables are put into the Builder constructor so that they can not be left out.

The Builder also includes a method called `build()` which returns the object that it is in and must be called at the end of the chain of method calls when building the object.

Following from the previous example, this code uses the Builder pattern for the Computer class.

```
public class Computer {

    private GraphicsCard graphicsCard;
    private Monitor[] monitors;
    private Processor processor;
    private Memory[] ram;
    //more class variables here...

    private Computer(Builder builder) {
        this.graphicsCard = builder.graphicsCard;
        this.monitors = builder.monitors;
        this.processor = builder.processor;
        this.ram = builder.ram;
    }

    public GraphicsCard getGraphicsCard() {
        return this.graphicsCard;
    }

    public Monitor[] getMonitors() {
        return this.monitors;
    }

    public Processor getProcessor() {
        return this.processor;
    }

    public Memory[] getRam() {
        return this.ram;
    }

    public static class Builder {
        private GraphicsCard graphicsCard;
        private Monitor[] monitors;
        private Processor processor;
        private Memory[] ram;

        public Builder(Processor p){
            this.processor = p;
        }

        public Builder graphicsCard(GraphicsCard g) {
            this.graphicsCard = g;
            return this;
        }

        public Builder monitors(Monitor[] mg) {
            this.monitors = mg;
            return this;
        }

        public Builder ram(Memory[] ram) {
            this.ram = ram;
            return this;
        }

        public Computer build() {
            return new Computer(this);
        }
```

```
    }
}
```

An example of how this class would be used:

```
public class ComputerExample {

    public static void main(String[] args) {
        Computer headlessComputer = new Computer.Builder(new Processor("Intel-i3"))
                .graphicsCard(new GraphicsCard("GTX-960"))
                .build();

        Computer gamingPC = new Computer.Builder(new Processor("Intel-i7-quadcode"))
                .graphicsCard(new GraphicsCard("DX11"))
                .monitors(new Monitor[] = {new Monitor("acer-s7"), new Monitor("acer-s7")})
                .ram(new Memory[] = {new Memory("2GB"), new Memory("2GB"), new Memory("2GB"),
new Memory("2GB")})
                .build();
    }

}
```

This example shows how the builder pattern can allow a lot of flexibility in how a class is created with fairly little effort. The Computer object can be implemented based on the callers desired configuration in an easy to read manner with little effort.

Builder pattern in Java with composition

Intent:

Separate the construction of a complex object from its representation so that the same construction process can create different representations

Builder pattern is useful when you have few mandatory attributes and many optional attributes to construct a object. To create an object with different mandatory and optional attributes, you have to provide complex constructor to create the object.Builder pattern provides simple step-by-step process to construct a complex object.

Real life use case:

Different users in FaceBook have different attributes, which consists of mandatory attribute like user name and optional attributes like UserBasicInfo and ContactInfo. Some users simply provide basic info. Some users provide detailed information including Contact info. In absence of Builder pattern, you have to provide a constructor with all mandatory and optional parameters. But Builder pattern simplifies the construction process by providing simple step-by-step process to construct the complex object.

Tips:

1. Provide a static nested builder class.
2. Provide constructor for Mandatory attributes of object.
3. Provide setter and getter methods for optional attributes of object.

4. Return the same Builder object after setting optional attributes.
5. Provide build() method, which returns complex object

Code snippet:

```
import java.util.*;

class UserBasicInfo{
    String nickName;
    String birthDate;
    String gender;

    public UserBasicInfo(String name,String date,String gender){
        this.nickName = name;
        this.birthDate = date;
        this.gender = gender;
    }

    public String toString(){
        StringBuilder sb = new StringBuilder();

sb.append("Name:DOB:Gender:").append(nickName).append(":").append(birthDate).append(":").
        append(gender);
        return sb.toString();
    }
}

class ContactInfo{
    String eMail;
    String mobileHome;
    String mobileWork;

    public ContactInfo(String mail, String homeNo, String mobileOff){
        this.eMail = mail;
        this.mobileHome = homeNo;
        this.mobileWork = mobileOff;
    }
    public String toString(){
        StringBuilder sb = new StringBuilder();

sb.append("email:mobile(H):mobile(W):").append(eMail).append(":").append(mobileHome).append(":").appenc

        return sb.toString();
    }
}
class FaceBookUser {
    String userName;
    UserBasicInfo userInfo;
    ContactInfo contactInfo;

    public FaceBookUser(String uName){
        this.userName = uName;
    }
    public void setUserBasicInfo(UserBasicInfo info){
        this.userInfo = info;
    }
    public void setContactInfo(ContactInfo info){
        this.contactInfo = info;
    }
    public String getUserName(){
```

```
        return userName;
    }
    public UserBasicInfo getUserBasicInfo(){
        return userInfo;
    }
    public ContactInfo getContactInfo(){
        return contactInfo;
    }

    public String toString(){
        StringBuilder sb = new StringBuilder();

sb.append("|User|").append(userName).append("|UserInfo|").append(userInfo).append("|ContactInfo|").appe

        return sb.toString();
    }

    static class FaceBookUserBuilder{
        FaceBookUser user;
        public FaceBookUserBuilder(String userName){
            this.user = new FaceBookUser(userName);
        }
        public FaceBookUserBuilder setUserBasicInfo(UserBasicInfo info){
            user.setUserBasicInfo(info);
            return this;
        }
        public FaceBookUserBuilder setContactInfo(ContactInfo info){
            user.setContactInfo(info);
            return this;
        }
        public FaceBookUser build(){
            return user;
        }
    }
}
public class BuilderPattern{
    public static void main(String args[]){
        FaceBookUser fbUser1 = new FaceBookUser.FaceBookUserBuilder("Ravindra").build(); //
Mandatory parameters
        UserBasicInfo info = new UserBasicInfo("sunrise","25-May-1975","M");

        // Build User name + Optional Basic Info
        FaceBookUser fbUser2 = new FaceBookUser.FaceBookUserBuilder("Ravindra").
                                            setUserBasicInfo(info).build();

        // Build User name + Optional Basic Info + Optional Contact Info
        ContactInfo cInfo = new ContactInfo("xxx@xyz.com","1111111111","2222222222");
        FaceBookUser fbUser3 = new FaceBookUser.FaceBookUserBuilder("Ravindra").
                                            setUserBasicInfo(info).
                                            setContactInfo(cInfo).build();

        System.out.println("Facebook user 1:"+fbUser1);
        System.out.println("Facebook user 2:"+fbUser2);
        System.out.println("Facebook user 3:"+fbUser3);
    }
}
```

output:

```
Facebook user 1:|User|Ravindra|UserInfo|null|ContactInfo|null
```

```
Facebook user 2:|User|Ravindra|UserInfo|Name:DOB:Gender:sunrise:25-May-1975:M|ContactInfo|null
Facebook user 3:|User|Ravindra|UserInfo|Name:DOB:Gender:sunrise:25-May-
1975:M|ContactInfo|email:mobile(H):mobile(W):xxx@xyz.com:1111111111:2222222222
```

Explanation:

1. `FaceBookUser` is a complex object with below attributes using composition:

   ```
   String userName;
   UserBasicInfo userInfo;
   ContactInfo contactInfo;
   ```

2. `FaceBookUserBuilder` is a static builder class, which contains and builds `FaceBookUser`.

3. `userName` is only Mandatory parameter to build `FaceBookUser`

4. `FaceBookUserBuilder` builds `FaceBookUser` by setting optional parameters : `UserBasicInfo` and `ContactInfo`

5. This example illustrates three different FaceBookUsers with different attributes, built from Builder.

   1. fbUser1 was built as FaceBookUser with userName attribute only
   2. fbUser2 was built as FaceBookUser with userName and UserBasicInfo
   3. fbUser3 was built as FaceBookUser with userName,UserBasicInfo and ContactInfo

In above example, composition has been used instead of duplicating all attributes of FaceBookUser in Builder class.

In creational patterns, we will first start with simple pattern like `FactoryMethod` and move towards more flexible and complex patterns like `AbstractFactory` and `Builder`.

Java / Lombok

```
import lombok.Builder;

@Builder
public class Email {

    private String to;
    private String from;
    private String subject;
    private String body;

}
```

Usage example:

```
Email.builder().to("email1@email.com")
        .from("email2@email.com")
        .subject("Email subject")
        .body("Email content")
```

```
        .build();
```

Advanced Builder Pattern With Java 8 Lambda Expression

```
public class Person {
private final String salutation;
private final String firstName;
private final String middleName;
private final String lastName;
private final String suffix;
private final Address address;
private final boolean isFemale;
private final boolean isEmployed;
private final boolean isHomewOwner;

public Person(String salutation, String firstName, String middleName, String lastName, String
suffix, Address address, boolean isFemale, boolean isEmployed, boolean isHomewOwner) {
    this.salutation = salutation;
    this.firstName = firstName;
    this.middleName = middleName;
    this.lastName = lastName;
    this.suffix = suffix;
    this.address = address;
    this.isFemale = isFemale;
    this.isEmployed = isEmployed;
    this.isHomewOwner = isHomewOwner;
 }
}
```

Old way

```
public class PersonBuilder {
private String salutation;
private String firstName;
private String middleName;
private String lastName;
private String suffix;
private Address address;
private boolean isFemale;
private boolean isEmployed;
private boolean isHomewOwner;

public PersonBuilder withSalutation(String salutation) {
    this.salutation = salutation;
    return this;
}

public PersonBuilder withFirstName(String firstName) {
    this.firstName = firstName;
    return this;
}

public PersonBuilder withMiddleName(String middleName) {
    this.middleName = middleName;
    return this;
}

public PersonBuilder withLastName(String lastName) {
```

```
    this.lastName = lastName;
    return this;
}

public PersonBuilder withSuffix(String suffix) {
    this.suffix = suffix;
    return this;
}

public PersonBuilder withAddress(Address address) {
    this.address = address;
    return this;
}

public PersonBuilder withIsFemale(boolean isFemale) {
    this.isFemale = isFemale;
    return this;
}

public PersonBuilder withIsEmployed(boolean isEmployed) {
    this.isEmployed = isEmployed;
    return this;
}

public PersonBuilder withIsHomewOwner(boolean isHomewOwner) {
    this.isHomewOwner = isHomewOwner;
    return this;
}

public Person createPerson() {
    return new Person(salutation, firstName, middleName, lastName, suffix, address, isFemale,
isEmployed, isHomewOwner);
}
```

Advanced Way:

```
public class PersonBuilder {
public String salutation;
public String firstName;
public String middleName;
public String lastName;
public String suffix;
public Address address;
public boolean isFemale;
public boolean isEmployed;
public boolean isHomewOwner;

public PersonBuilder with(
        Consumer<PersonBuilder> builderFunction) {
    builderFunction.accept(this);
    return this;
}


public Person createPerson() {
    return new Person(salutation, firstName, middleName,
            lastName, suffix, address, isFemale,
            isEmployed, isHomewOwner);
}
```

}

Usage:

```
Person person = new PersonBuilder()
    .with($ -> {
        $.salutation = "Mr.";
        $.firstName = "John";
        $.lastName = "Doe";
        $.isFemale = false;
        $.isHomewOwner = true;
        $.address =
            new PersonBuilder.AddressBuilder()
                .with($_address -> {
                    $_address.city = "Pune";
                    $_address.state = "MH";
                    $_address.pin = "411001";
                }).createAddress();
    })
    .createPerson();
```

Refer: https://medium.com/beingprofessional/think-functional-advanced-builder-pattern-using-lambda-284714b85ed5#.d9sryx3g9

Read Builder Pattern online: https://riptutorial.com/design-patterns/topic/1811/builder-pattern

Chapter 6: Chain of Responsibility

Examples

Chain of Responsibility example (Php)

A method called in one object will move up the chain of objects until one is found that can properly handle the call. This particular example uses scientific experiments with functions that can just get the title of the experiment, the experiments id or the tissue used in the experiment.

```
abstract class AbstractExperiment {
    abstract function getExperiment();
    abstract function getTitle();
}

class Experiment extends AbstractExperiment {
    private $experiment;
    private $tissue;
    function __construct($experiment_in) {
        $this->experiment = $experiment_in;
        $this->tissue = NULL;
    }
    function getExperiment() {
        return $this->experiment;
    }
    //this is the end of the chain - returns title or says there is none
    function getTissue() {
      if (NULL != $this->tissue) {
        return $this->tissue;
      } else {
        return 'there is no tissue applied';
      }
    }
}

class SubExperiment extends AbstractExperiment {
    private $experiment;
    private $parentExperiment;
    private $tissue;
    function __construct($experiment_in, Experiment $parentExperiment_in) {
      $this->experiment = $experiment_in;
      $this->parentExperiment = $parentExperiment_in;
      $this->tissue = NULL;
    }
    function getExperiment() {
      return $this->experiment;
    }
    function getParentExperiment() {
      return $this->parentExperiment;
    }
    function getTissue() {
      if (NULL != $this->tissue) {
        return $this->tissue;
      } else {
        return $this->parentExperiment->getTissue();
      }
```

```
    }
}

//This class and all further sub classes work in the same way as SubExperiment above
class SubSubExperiment extends AbstractExperiment {
    private $experiment;
    private $parentExperiment;
    private $tissue;
    function __construct($experiment_in, Experiment $parentExperiment_in) { //as above }
    function getExperiment() { //same as above }
    function getParentExperiment() { //same as above }
    function getTissue() { //same as above }
}
```

Read Chain of Responsibility online: https://riptutorial.com/design-patterns/topic/6083/chain-of-responsibility

Chapter 7: Command pattern

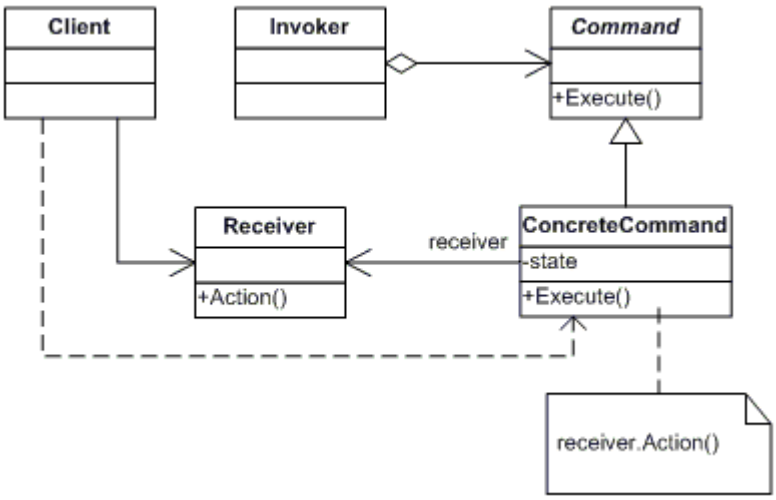
Examples

Command pattern example in Java

wikipedia definition:

> Command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time

UML diagram from dofactory:



Basic components and workflow:

1. `Command` declares an interface for abstract commands like `execute()`
2. `Receiver` knows how to execute a particular command
3. `Invoker` holds `ConcreteCommand`, which has to be executed
4. `Client` creates `ConcreteCommand` and assign `Receiver`
5. `ConcreteCommand` defines binding between `Command` and `Receiver`

In this way, Command pattern decouples **Sender** (Client) from **Receiver** through **Invoker**. **Invoker** has complete knowledge of which **Command** to be executed and **Command** knows which **Receiver** to be invoked to execute a particular operation.

Code snippet:

```
interface Command {
    void execute();
}
class Receiver {
    public void switchOn(){
        System.out.println("Switch on from:"+this.getClass().getSimpleName());
    }
```

```
}
class OnCommand implements Command{
    private Receiver receiver;

    public OnCommand(Receiver receiver){
        this.receiver = receiver;
    }
    public void execute(){
        receiver.switchOn();
    }
}
class Invoker {
    private Command command;

    public Invoker(Command command){
        this.command = command;
    }
    public void execute(){
        this.command.execute();
    }
}

class TV extends Receiver{

    public String toString(){
        return this.getClass().getSimpleName();
    }
}
class DVDPlayer extends Receiver{

    public String toString(){
        return this.getClass().getSimpleName();
    }
}

public class CommandDemoEx{
    public static void main(String args[]){
        // On command for TV with same invoker
        Receiver receiver = new TV();
        Command onCommand = new OnCommand(receiver);
        Invoker invoker = new Invoker(onCommand);
        invoker.execute();

        // On command for DVDPlayer with same invoker
        receiver = new DVDPlayer();
        onCommand = new OnCommand(receiver);
        invoker = new Invoker(onCommand);
        invoker.execute();
    }
}
```

output:

```
Switch on from:TV
Switch on from:DVDPlayer
```

Explanation:

In this example,

1. **Command** interface defines `execute()` method.
2. **OnCommand** is **ConcreteCommand**, which implements `execute()` method.
3. **Receiver** is the base class.
4. **TV** and **DVDPlayer** are two types of **Receivers**, which are passed to ConcreteCommand like OnCommand.
5. **Invoker** contains **Command**. It's the key to de-couple Sender from **Receiver**.
6. **Invoker** receives **OnCommand** -> which calls **Receiver** (TV) to execute this command.

By using Invoker, you can switch on TV and DVDPlayer. If you extend this program, you switch off both TV and DVDPlayer too.

Key use cases:

1. To implement callback mechanism
2. To implement undo and redo functionality
3. To Maintain a history of commands

Read Command pattern online: https://riptutorial.com/design-patterns/topic/2677/command-pattern

Chapter 8: Composite pattern

Examples

Composite logger

The Composite pattern is a design pattern that allows to treat a group of objects as a single instance of an object. It is one of the Gang of Four's structural design patterns.

Example below demonstrate how Composite can be used to log to multiple places using single Log invocation. This approach adheres to SOLID principles because it allows you to add new logging mechanism without violating Single responsibility principle (each logger has only one responsibility) or Open/closed principle (You can add new logger that will log to new place by adding new implementation and not by modifying existing ones).

```
public interface ILogger
{
    void Log(string message);
}

public class CompositeLogger : ILogger
{
    private readonly ILogger[] _loggers;

    public CompositeLogger(params ILogger[] loggers)
    {
        _loggers = loggers;
    }

    public void Log(string message)
    {
        foreach (var logger in _loggers)
        {
            logger.Log(message);
        }
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        //log to console
    }
}

public class FileLogger : ILogger
{
    public void Log(string message)
    {
        //log to file
    }
}
```

```
var compositeLogger = new CompositeLogger(new ConsoleLogger(), new FileLogger());
compositeLogger.Log("some message"); //this will be invoked both on ConsoleLogger and
FileLogger
```

It is worth mentioning that composite loggers can be nested (one of the parameters to composite loggers constructor can be another composite logger) creating tree-like structure.

Read Composite pattern online: https://riptutorial.com/design-patterns/topic/4515/composite-pattern

Chapter 9: Composite pattern

Introduction

Composite lets clients treat individual objects and compositions of objects uniformly. For example consider a program that manipulates a file system. Files are simple objects and folders are composition of files and folders. However, for example, they both have size, name etc. functions. It would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface

Remarks

The composite pattern applies when there is a part-whole hierarchy of objects and a client needs to deal with objects uniformly regardless of the fact that an object might be a leaf (simple object) or a branch (composite object).

Examples

pseudocode for a dumb file manager

```
/*
* Component is an interface
* which all elements (files,
* folders, links ...) will implement
*/
class Component
{
public:
    virtual int getSize() const = 0;
};

/*
* File class represents a file
* in file system.
*/
class File : public Component
{
public:
    virtual int getSize() const {
        // return file size
    }
};

/*
* Folder is a component and
* also may contain files and
* another folders. Folder is a
* composition of components
*/
class Folder : public Component
{
```

```
public:
    void addComponent(Component* aComponent) {
        // mList append aComponent;
    }
    void removeComponent(Component* aComponent) {
        // remove aComponent from mList
    }
    virtual int getSize() const {
        int size = 0;
        foreach(component : mList) {
            size += component->getSize();
        }
        return size;
    }

private:
    list<Component*> mList;
};
```

Read Composite pattern online: https://riptutorial.com/design-patterns/topic/9197/composite-pattern

Chapter 10: Data Access Object(DAO) design pattern

Examples

Data Access Object J2EE design pattern with Java

Data Access Object(DAO) design pattern is a standard J2EE design pattern.

In this design pattern data is accessed through classes containing methods to access data from databases or other sources, which are called *data access objects*. Standard practice assumes that there are POJO classes. DAO can be mixed with other Design Patterns to access data, such as with MVC(model view controller), Command Patterns etc.

The following is an example of DAO design pattern. It has an **Employee** class, a DAO for Employee called **EmployeeDAO** and an **ApplicationView** class to demonstrate the examples.

Employee.java

```
public class Employee {
    private Integer employeeId;
    private String firstName;
    private String lastName;
    private Integer salary;

    public Employee(){

    }

    public Employee(Integer employeeId, String firstName, String lastName, Integer salary) {
        super();
        this.employeeId = employeeId;
        this.firstName = firstName;
        this.lastName = lastName;
        this.setSalary(salary);
    }

    //standard setters and getters

}
```

EmployeeDAO

```
public class EmployeeDAO {

    private List<Employee> employeeList;

    public EmployeeDAO(List<Employee> employeeList){
        this.employeeList = employeeList;
    }
```

```
    public List<Employee> getAllEmployees(){
        return employeeList;
    }

    //add other retrieval methods as you wish
    public Employee getEmployeeWithMaxSalary(){
        Employee employee = employeeList.get(0);
        for (int i = 0; i < employeeList.size(); i++){
            Employee e = employeeList.get(i);
            if (e.getSalary() > employee.getSalary()){
                employee = e;
            }
        }

        return employee;
    }

}
```

ApplicationView.java

```
public class ApplicationView {

    public static void main(String[] args) {
        // See all the employees with data access object

        List<Employee> employeeList = setEmployeeList();
        EmployeeDAO eDAO = new EmployeeDAO(employeeList);

        List<Employee> allEmployees = eDAO.getAllEmployees();

        for (int i = 0; i < allEmployees.size(); i++) {
            Employee e = employeeList.get(i);
            System.out.println("UserId: " + e.getEmployeeId());
        }

        Employee employeeWithMaxSalary = eDAO.getEmployeeWithMaxSalary();

        System.out.println("Maximum Salaried Employee" + " FirstName:" +
employeeWithMaxSalary.getFirstName()
        + " LastName:" + employeeWithMaxSalary.getLastName() + " Salary: " +
employeeWithMaxSalary.getSalary());

    }

    public static List<Employee> setEmployeeList() {
        Employee employee1 = new Employee(1, "Pete", "Samprus", 3000);
        Employee employee2 = new Employee(2, "Peter", "Russell", 4000);
        Employee employee3 = new Employee(3, "Shane", "Watson", 2000);

        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(employee1);
        employeeList.add(employee2);
        employeeList.add(employee3);
        return employeeList;
    }

}
```

Hence we have an example where we see how to use Data Access Object design pattern.

Read Data Access Object(DAO) design pattern online: https://riptutorial.com/design-patterns/topic/6351/data-access-object-dao--design-pattern

Chapter 11: Decorator pattern

Introduction

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

Parameters

Parameter	Description
Beverage	it can be Tea or Coffee

Examples

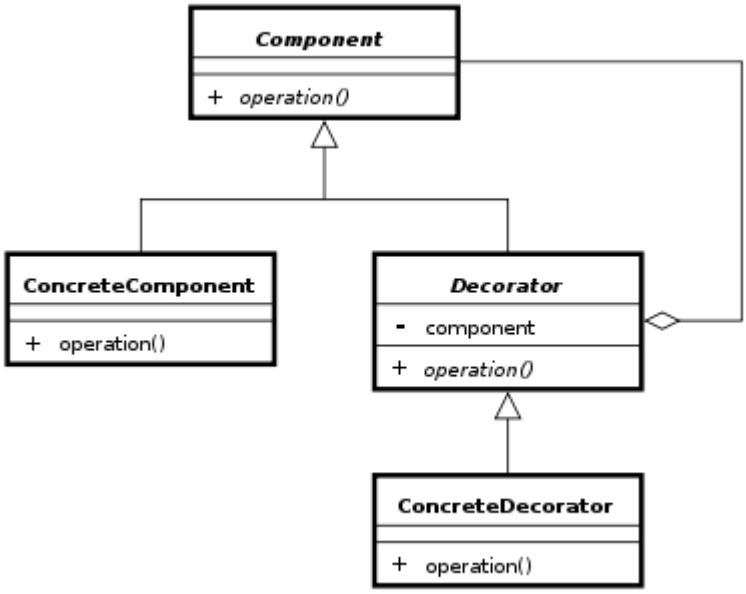
VendingMachineDecorator

Definition of Decorator as per Wikiepdia:

The Decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time, independently of other instances of the same class, provided some groundwork is done at design time.

Decorator attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator pattern contains four components.



1. Component Interface: It defines an interface to execute particular operations
2. ConcreteComponent: It implements the operations defined in Component interface
3. Decorator (Abstract) : it is an abstract class, which extends the component interface. It contains Component interface. In absence of this class, you need many sub-classes of ConcreteDecorators for different combinations. Composition of component reduces un-necessary sub-classes.
4. ConcreteDecorator: It holds the implementation of Abstract Decorator.

Coming back to example code,

1. *Beverage* is component. It defines an abstract method: decorateBeverage
2. *Tea* and *Coffee* are concrete implementations of *Beverage*.
3. *BeverageDecorator* is an abstract class, which contains *Beverage*
4. SugarDecorator and LemonDecorator are concrete Decorators for BeverageDecorator.

EDIT: Changed the example to reflect real world scenario of computing the price of Beverage by adding one or more flavours like Sugar, Lemon etc(flavours are decorators)

```
abstract class Beverage {
    protected String name;
    protected int price;
    public Beverage(){

    }
    public  Beverage(String name){
        this.name = name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    protected void setPrice(int price){
        this.price = price;
    }
```

```
    protected int getPrice(){
        return price;
    }
    protected abstract void decorateBeverage();

}
class Tea extends Beverage{
    public Tea(String name){
        super(name);
        setPrice(10);
    }
    public void decorateBeverage(){
        System.out.println("Cost of:"+ name +":"+ price);
        // You can add some more functionality
    }
}
class Coffee extends Beverage{
    public Coffee(String name){
        super(name);
        setPrice(15);
    }
    public void decorateBeverage(){
        System.out.println("Cost of:"+ name +":"+ price);
        // You can add some more functionality
    }
}
abstract class BeverageDecorator extends Beverage {
    protected Beverage beverage;
    public BeverageDecorator(Beverage beverage){
        this.beverage = beverage;
        setName(beverage.getName()+"+"+getDecoratedName());
        setPrice(beverage.getPrice()+getIncrementPrice());
    }
    public void decorateBeverage(){
        beverage.decorateBeverage();
        System.out.println("Cost of:"+getName()+":"+getPrice());
    }
    public abstract int getIncrementPrice();
    public abstract String getDecoratedName();
}
class SugarDecorator extends BeverageDecorator{
    public SugarDecorator(Beverage beverage){
        super(beverage);
    }
    public void decorateBeverage(){
        super.decorateBeverage();
        decorateSugar();
    }
    public void decorateSugar(){
        System.out.println("Added Sugar to:"+beverage.getName());
    }
    public int getIncrementPrice(){
        return 5;
    }
    public String getDecoratedName(){
        return "Sugar";
    }
}
class LemonDecorator extends BeverageDecorator{
    public LemonDecorator(Beverage beverage){
        super(beverage);
```

```
    }
    public void decorateBeverage(){
        super.decorateBeverage();
        decorateLemon();
    }
    public void decorateLemon(){
        System.out.println("Added Lemon to:"+beverage.getName());
    }
    public int getIncrementPrice(){
        return 3;
    }
    public String getDecoratedName(){
        return "Lemon";
    }
}

public class VendingMachineDecorator {
    public static void main(String args[]){
        Beverage beverage = new SugarDecorator(new LemonDecorator(new Tea("Assam Tea")));
        beverage.decorateBeverage();
        beverage = new SugarDecorator(new LemonDecorator(new Coffee("Cappuccino")));
        beverage.decorateBeverage();
    }
}
```

output:

```
Cost of:Assam Tea:10
Cost of:Assam Tea+Lemon:13
Added Lemon to:Assam Tea
Cost of:Assam Tea+Lemon+Sugar:18
Added Sugar to:Assam Tea+Lemon
Cost of:Cappuccino:15
Cost of:Cappuccino+Lemon:18
Added Lemon to:Cappuccino
Cost of:Cappuccino+Lemon+Sugar:23
Added Sugar to:Cappuccino+Lemon
```

This example computes cost of beverage in Vending Machine after adding many flavours to the beverage.

In above example:

Cost of Tea = 10, Lemon = 3 and Sugar = 5. If you make Sugar + Lemon + Tea, it costs 18.

Cost of Coffee =15, Lemon = 3 and Sugar = 5. If you make Sugar + Lemon + Coffee, it costs 23

By using same Decorator for both beverages (Tea and Coffee), the number of sub-classes have been reduced. In absence of Decorator pattern, you should have different sub classes for different combinations.

The combinations will be like this:

```
SugarLemonTea
SugarTea
LemonTea
```

```
SugarLemonCapaccuino
SugarCapaccuino
LemonCapaccuino
```

etc.

By using same `Decorator` for both beverages, the number of sub-classes have been reduced. It's possible due to `composition` rather than `inheritance` concept used in this pattern.

Comparison with other Design patterns (From sourcemaking article)

1. *Adapter* provides a different interface to its subject. *Proxy* provides the same interface. *Decorator* provides an enhanced interface.
2. *Adapter* changes an object's interface, *Decorator* enhances an object's responsibilities.
3. *Composite* and *Decorator* have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects
4. *Decorator* is designed to let you add responsibilities to objects without subclassing. *Composite's* focus is not on embellishment but on representation
5. *Decorator* and *Proxy* have different purposes but similar structures
6. *Decorator* lets you change the skin of an object. *Strategy* lets you change the guts.

Key use cases:

1. Add additional functionalities/responsibilities dynamically
2. Remove functionalities/responsibilities dynamically
3. Avoid too much of sub-classing to add additional responsibilities.

Caching Decorator

This example demonstrate how to add caching capabilities to `DbProductRepository` using Decorator pattern. This approach adheres to SOLID principles because it allows you to add caching without violating Single responsibility principle or Open/closed principle.

```
public interface IProductRepository
{
    Product GetProduct(int id);
}

public class DbProductRepository : IProductRepository
{
    public Product GetProduct(int id)
    {
        //return Product retrieved from DB
    }
}

public class ProductRepositoryCachingDecorator : IProductRepository
```

```
{
    private readonly IProductRepository _decoratedRepository;
    private readonly ICache _cache;
    private const int ExpirationInHours = 1;

    public ProductRepositoryCachingDecorator(IProductRepository decoratedRepository, ICache
cache)
    {
        _decoratedRepository = decoratedRepository;
        _cache = cache;
    }

    public Product GetProduct(int id)
    {
        var cacheKey = GetKey(id);
        var product = _cache.Get<Product>(cacheKey);
        if (product == null)
        {
            product = _decoratedRepository.GetProduct(id);
            _cache.Set(cacheKey, product, DateTimeOffset.Now.AddHours(ExpirationInHours));
        }

        return product;
    }

    private string GetKey(int id) => "Product:" + id.ToString();
}

public interface ICache
{
    T Get<T>(string key);
    void Set(string key, object value, DateTimeOffset expirationTime)
}
```

Usage:

```
var productRepository = new ProductRepositoryCachingDecorator(new DbProductRepository(), new
Cache());
var product = productRepository.GetProduct(1);
```

Result of invoking `GetProduct` will be: retrieve product from cache (decorator responsibility), if product was not in the cache proceed with invocation to `DbProductRepository`and retrieve product from DB. After this product can be added to the cache so subsequent calls won't hit DB.

Read Decorator pattern online: https://riptutorial.com/design-patterns/topic/1720/decorator-pattern

Chapter 12: Dependency Injection

Introduction

The general idea behind Dependency Injection is that you design your application around loosely coupled components while adhering to the Dependency Inversion Principle. By not depending on concrete implementations, allows to design highly flexible systems.

Remarks

The basic idea behind dependency injection is to create more loosely coupled code. When a class, rather than newing up its own dependencies, takes in its dependencies instead, the class becomes more simple to test as a unit (unit testing).

To further elaborate on loose coupling - the idea is that classes become dependent on abstractions, rather than concretions. If class `A` depends on another concrete class `B`, then there is no real testing of `A` without `B`. While this sort of test can be OK, it does not lend itself to unit testable code. A loosely coupled design would define an abstraction `IB` (as an example) that class `A` would depend on. `IB` can then be mocked to provide testable behavior, rather than relying on the real implementation of `B` to be able to provide testable scenarios to `A`.

Tightly coupled example (C#):

```
public class A
{
    public void DoStuff()
    {
        B b = new B();
        b.Foo();
    }
}
```

In the above, class `A` depends on `B`. There is no testing `A` without the concrete `B`. While this is fine in an integration testing scenario, it is difficult to unit test `A`.

A more loosely coupled implementation of the above could look like:

```
public interface IB
{
    void Foo();
}

public class A
{
    private readonly IB _iB;

    public A(IB iB)
    {
        _iB = iB;
    }
```

```
    public void DoStuff()
    {
        _b.Foo();
    }
}
```

The two implementations seem quite similar, there is however an important difference. Class `A` is no longer directly dependent on class `B`, it is now dependent on `IB`. Class `A` no longer has the responsibility of newing up its own depedencies - they must now be provided ***to*** `A`.

Examples

Setter injection (C#)

```
public class Foo
{
    private IBar _iBar;
    public IBar iBar { set { _iBar = value; } }

    public void DoStuff()
    {
        _iBar.DoSomething();
    }
}

public interface IBar
{
    void DoSomething();
}
```

Constructor Injection (C#)

```
public class Foo
{
    private readonly IBar _iBar;

    public Foo(IBar iBar)
    {
        _iBar = iBar;
    }

    public void DoStuff()
    {
        _bar.DoSomething();
    }
}

public interface IBar
{
    void DoSomething();
}
```

Read Dependency Injection online: https://riptutorial.com/design-patterns/topic/1723/dependency-

injection

Chapter 13: Facade

Examples

Real world facade (C#)

```
public class MyDataExporterToExcell
{
    public static void Main()
    {
        GetAndExportExcelFacade facade = new GetAndExportExcelFacade();

        facade.Execute();
    }
}

public class GetAndExportExcelFacade
{

    // All services below do something by themselves, determine location for data,
    // get the data, format the data, and export the data
    private readonly DetermineExportDatabaseService _determineExportData = new
DetermineExportDatabaseService();
    private readonly GetRawDataToExportFromDbService _getRawData = new
GetRawDataToExportFromDbService();
    private readonly TransformRawDataForExcelService _transformData = new
TransformRawDataForExcelService();
    private readonly CreateExcelExportService _createExcel = new CreateExcelExportService();

    // the facade puts all the individual pieces together, as its single responsibility.
    public void Execute()
    {
        var dataLocationForExport = _determineExportData.GetDataLocation();
        var rawData = _getRawData.GetDataFromDb(dataLocationForExport);
        var transformedData = _transformData.TransformRawToExportableObject(rawData);
        _createExcel.GenerateExcel("myFilename.xlsx");
    }
}
```

Facade example in java

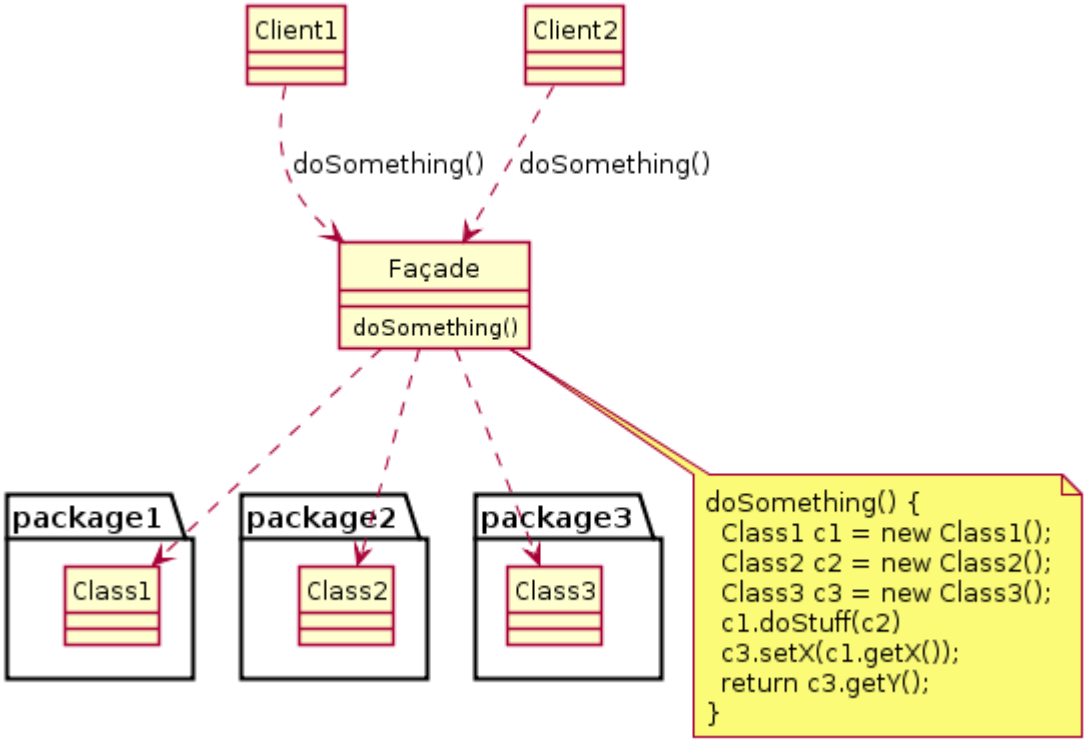
Facade is structural design pattern. It hides the complexities of large system and provides a simple interface to client.

Client uses only **Facade** and it's not worried about inter dependencies of sub-systems.

Definition from Gang of Four book:

> Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use

Structure:



Real world example:

Think of some travel booking sites like makemytrip, cleartrip which offers services to book Trains, Flights and Hotels.

Code snippet:

```
import java.util.*;

public class TravelFacade{
    FlightBooking flightBooking;
    TrainBooking trainBooking;
    HotelBooking hotelBooking;

    enum BookingType {
        Flight,Train,Hotel,Flight_And_Hotel,Train_And_Hotel;
    };

    public TravelFacade(){
        flightBooking = new FlightBooking();
        trainBooking = new TrainBooking();
        hotelBooking = new HotelBooking();
    }
    public void book(BookingType type, BookingInfo info){
        switch(type){
            case Flight:
                // book flight;
                flightBooking.bookFlight(info);
                return;
            case Hotel:
                // book hotel;
                hotelBooking.bookHotel(info);
                return;
            case Train:
                // book Train;
```

```
                trainBooking.bookTrain(info);
                return;
            case Flight_And_Hotel:
                // book Flight and Hotel
                flightBooking.bookFlight(info);
                hotelBooking.bookHotel(info);
                return;
             case Train_And_Hotel:
                // book Train and Hotel
                trainBooking.bookTrain(info);
                hotelBooking.bookHotel(info);
                return;
        }
    }
}
class BookingInfo{
    String source;
    String destination;
    Date    fromDate;
    Date     toDate;
    List<PersonInfo> list;
}
class PersonInfo{
    String name;
    int       age;
    Address address;
}
class Address{

}
class FlightBooking{
    public FlightBooking(){

    }
    public void bookFlight(BookingInfo info){

    }
}
class HotelBooking{
    public HotelBooking(){

    }
    public void bookHotel(BookingInfo info){

    }
}
class TrainBooking{
    public TrainBooking(){

    }
    public void bookTrain(BookingInfo info){

    }
}
```

Explanation:

1. `FlightBooking, TrainBooking and HotelBooking` are different sub-systems of large system : `TravelFacade`

2. `TravelFacade` offers a simple interface to book one of below options

```
Flight Booking
Train Booking
Hotel Booking
Flight + Hotel booking
Train + Hotel booking
```

3. book API from TravelFacade internally calls below APIs of sub-systems

```
flightBooking.bookFlight
trainBooking.bookTrain(info);
hotelBooking.bookHotel(info);
```

4. In this way, `TravelFacade` provides simpler and easier API with-out exposing sub-system APIs.

Applicability and Use cases (from Wikipedia) :

1. A simple interface is required to access a complex system.
2. The abstractions and implementations of a subsystem are tightly coupled.
3. Need an entry point to each level of layered software.
4. System is very complex or difficult to understand.

Read Facade online: https://riptutorial.com/design-patterns/topic/3516/facade

Chapter 14: Factory

Remarks

> Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
>
> *-- GOF 1994*

Examples

Simple factory (Java)

A factory decreases coupling between code that needs to create objects from object creation code. Object creation is not made explicitly by calling a class constructor but by calling some function that creates the object on behalf the caller. A simple Java example is the following one:

```
interface Car {
}

public class CarFactory{
  static public Car create(String s) {
    switch (s) {
    default:
    case "us":
    case "american": return new Chrysler();
    case "de":
    case "german": return new Mercedes();
    case "jp":
    case "japanese": return new Mazda();
    }
  }
}

class Chrysler implements Car {
  public String toString() { return "Chrysler"; }
}

class Mazda implements Car {
  public String toString() { return "Mazda"; }
}

class Mercedes implements Car {
  public String toString() { return "Mercedes"; }
}

public class CarEx {
  public static void main(String args[]) {
    Car car = CarFactory.create("us");
    System.out.println(car);
  }
}
```

In this example, user just gives some hint about what he needs and the factory is free to construct something appropriate. It is a **dependency inversion**: the implementor of `Car` concept is free to return an appropriate concrete `Car` requested by the user which in turn doesn't know the details of the concrete object built.

This is a simple example of how factory works, of course in this example it is always possible to instantiate concrete classes; but one can prevent it by hiding concrete classes in a package, such that user is forced to use the factory.

.Net Fiddle for above example.

Abstract factory (C++)

Abstract factory pattern provides a way to obtain an coherent collection of objects through a collection of factories functions. As for every pattern, coupling is reduced by abstracting the way a set of objects are created, so that the user code is unaware of the many details of the objects he needs.

The following C++ example illustrates how to obtain different kind of objects of the same (hypothetical) GUI family:

```
#include <iostream>

/* Abstract definitions */
class GUIComponent {
public:
  virtual ~GUIComponent() = default;
  virtual void draw() const = 0;
};
class Frame  : public GUIComponent {};
class Button : public GUIComponent {};
class Label  : public GUIComponent {};

class GUIFactory {
public:
  virtual ~GUIFactory() = default;
  virtual std::unique_ptr<Frame> createFrame() = 0;
  virtual std::unique_ptr<Button> createButton() = 0;
  virtual std::unique_ptr<Label> createLabel() = 0;
  static std::unique_ptr<GUIFactory> create(const std::string& type);
};

/* Windows support */
class WindowsFactory : public GUIFactory {
private:
    class WindowsFrame : public Frame {
    public:
      void draw() const override { std::cout << "I'm a Windows-like frame" << std::endl; }
    };
    class WindowsButton : public Button {
    public:
      void draw() const override { std::cout << "I'm a Windows-like button" << std::endl; }
    };
    class WindowsLabel : public Label {
    public:
```

```
      void draw() const override { std::cout << "I'm a Windows-like label" << std::endl; }
    };
public:
  std::unique_ptr<Frame> createFrame() override { return std::make_unique<WindowsFrame>(); }
  std::unique_ptr<Button> createButton() override { return std::make_unique<WindowsButton>();
}
  std::unique_ptr<Label> createLabel() override { return std::make_unique<WindowsLabel>(); }
};

/* Linux support */
class LinuxFactory : public GUIFactory {
private:
    class LinuxFrame : public Frame {
    public:
      void draw() const override { std::cout << "I'm a Linux-like frame" << std::endl; }
    };
    class LinuxButton : public Button {
    public:
      void draw() const override { std::cout << "I'm a Linux-like button" << std::endl; }
    };
    class LinuxLabel : public Label {
    public:
      void draw() const override { std::cout << "I'm a Linux-like label" << std::endl; }
    };
public:
  std::unique_ptr<Frame> createFrame() override { return std::make_unique<LinuxFrame>(); }
  std::unique_ptr<Button> createButton() override { return std::make_unique<LinuxButton>(); }
  std::unique_ptr<Label> createLabel() override { return std::make_unique<LinuxLabel>(); }
};

std::unique_ptr<GUIFactory> GUIFactory::create(const string& type) {
  if (type == "windows") return std::make_unique<WindowsFactory>();
  return std::make_unique<LinuxFactory>();
}

/* User code */
void buildInterface(GUIFactory& factory) {
  auto frame = factory.createFrame();
  auto button = factory.createButton();
  auto label = factory.createLabel();

  frame->draw();
  button->draw();
  label->draw();
}

int main(int argc, char *argv[]) {
  if (argc < 2) return 1;
  auto guiFactory = GUIFactory::create(argv[1]);
  buildInterface(*guiFactory);
}
```

If the generated executable is named `abstractfactory` then output may give:

```
$ ./abstractfactory windows
I'm a Windows-like frame
I'm a Windows-like button
I'm a Windows-like label
$ ./abstractfactory linux
I'm a Linux-like frame
```

```
I'm a Linux-like button
I'm a Linux-like label
```

Simple example of Factory that uses an IoC (C#)

Factories can be used in conjunction with Inversion of Control (IoC) libraries too.

- The typical use case for such a factory is when we want to create an object based on parameters that are not known until run-time (such as the current User).
- In these cases it can be sometimes be difficult (if not impossible) to configure the IoC library alone to handle this kind of runtime contextual information, so we can wrap it in a factory.

Example

- Suppose we have a `User` class, whose characteristics (ID, Security clearance level, etc.), are unknown until runtime (since the current user could be anyone who uses the application).
- We need to take the current User and obtain an `ISecurityToken` for them, which can then be used to check if the user is allowed to perform certain actions or not.
- The implementation of ISecurityToken will vary depending on the level of the User - in other words, ISecurityToken uses *polymorphism*.

In this case, we have two implementations, which also use *Marker Interfaces* to make it easier to identify them to the IoC library; the IoC library in this case is just made up and identified by the abstraction `IContainer`.

> Note also that many modern IoC factories have native capabilities or plugins that allow auto-creation of factories as well as avoiding the need for marker interfaces as shown below; however since not all do, this example caters to a simple, lowest common functionality concept.

```
//describes the ability to allow or deny an action based on PerformAction.SecurityLevel
public interface ISecurityToken
{
    public bool IsAllowedTo(PerformAction action);
}

//Marker interface for Basic permissions
public interface IBasicToken:ISecurityToken{};
//Marker interface for super permissions
public interface ISuperToken:ISecurityToken{};

//since IBasictoken inherits ISecurityToken, BasicToken can be treated as an ISecurityToken
public class BasicToken:IBasicToken
{
     public bool IsAllowedTo(PerformAction action)
     {
         //Basic users can only perform basic actions
         if(action.SecurityLevel!=SecurityLevel.Basic) return false;
         return true;
     }
}

public class SuperToken:ISuperToken
```

```
{
     public bool IsAllowedTo(PerformAction action)
     {
         //Super users can perform all actions
         return true;
     }
}
```

Next we will create a `SecurityToken` factory, which will take as a dependency our `IContainer`

```
public class SecurityTokenFactory
{
   readonly IContainer _container;
   public SecurityTokenFactory(IContainer container)
   {
      if(container==null) throw new ArgumentNullException("container");
   }

   public ISecurityToken GetToken(User user)
   {
      if (user==null) throw new ArgumentNullException("user);
      //depending on the user security level, we return a different type; however all types
implement ISecurityToken so the factory can produce them.
      switch user.SecurityLevel
      {
          case Basic:
           return _container.GetInstance<BasicSecurityToken>();
          case SuperUser:
           return _container.GetInstance<SuperUserToken>();
      }
   }
}
```

Once we've registered these with the `IContainer`:

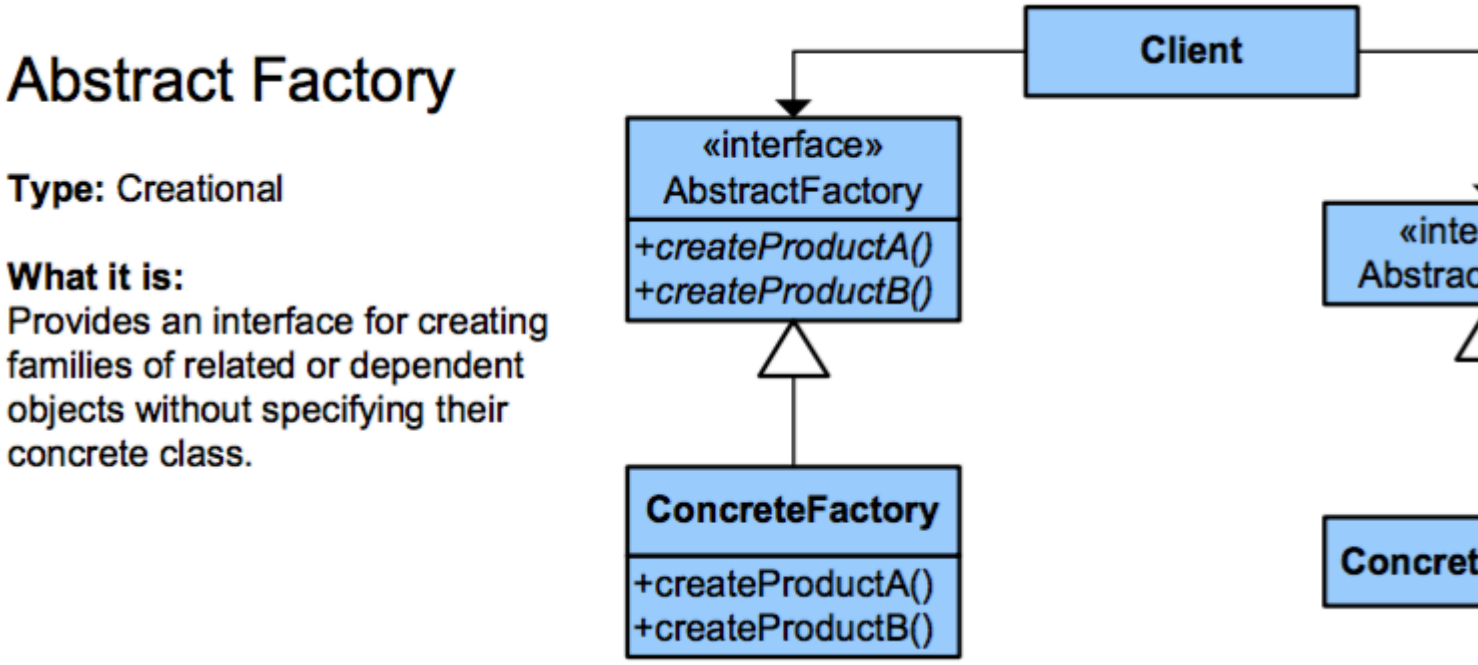
```
IContainer.For<SecurityTokenFactory>().Use<SecurityTokenFactory>().Singleton(); //we only need
a single instance per app
IContainer.For<IBasicToken>().Use<BasicToken>().PerRequest(); //we need an instance per-
request
IContainer.For<ISuperToken>().Use<SuperToken>().PerRequest();//we need an instance per-request
```

the consuming code can use it to get the correct token at runtime:

```
readonly SecurityTokenFactory _tokenFactory;
...
...
public void LogIn(User user)
{
    var token = _tokenFactory.GetToken(user);
    user.SetSecurityToken(token);
}
```

> In this way we benefit from the encapsulation provided by the factory and also from the lifecycle management provided by the IoC library.

An Abstract Factory



The following design pattern is categorized as a creational pattern.

An abstract factory is used to provide an interface for creating families of related objects, without specifying concrete classes and can be used to hide platform specific classes.

```
interface Tool {
    void use();
}

interface ToolFactory {
    Tool create();
}

class GardenTool implements Tool {

    @Override
    public void use() {
         // Do something...
    }
}

class GardenToolFactory implements ToolFactory {

    @Override
    public Tool create() {
        // Maybe additional logic to setup...
        return new GardenTool();
    }
}

class FarmTool implements Tool {

    @Override
    public void use() {
        // Do something...
```

```
    }
}

class FarmToolFactory implements ToolFactory {

    @Override
    public Tool create() {
        // Maybe additional logic to setup...
        return new FarmTool();
    }
}
```

Then a supplier/producer of some sort would be used which would be passed information that would allow it to give back the correct type of factory implementation:

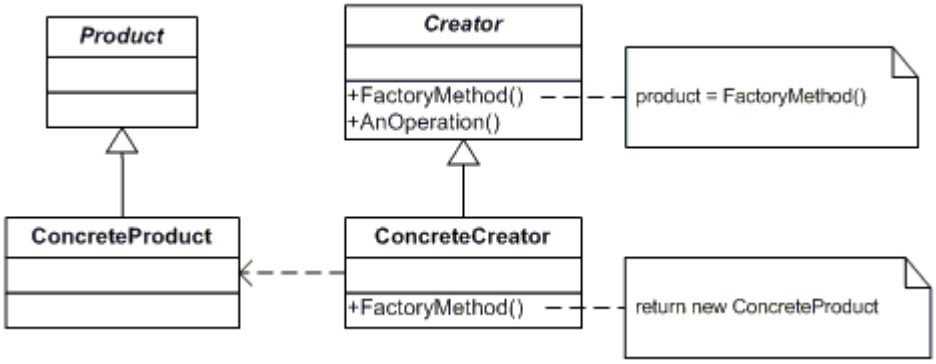
```
public final class FactorySupplier {

    // The supported types it can give you...
    public enum Type {
        FARM, GARDEN
    };

    private FactorySupplier() throws IllegalAccessException {
        throw new IllegalAccessException("Cannot be instantiated");
    }

    public static ToolFactory getFactory(Type type) {

        ToolFactory factory = null;

        switch (type) {
        case FARM:
            factory = new FarmToolFactory();
            break;
        case GARDEN:
            factory = new GardenToolFactory();
            break;
        } // Could potentially add a default case to handle someone passing in null

        return factory;
    }
}
```

Factory example by implementing Factory method (Java)

Intent:

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

UML diagram:



Product: It defines an interface of the objects the Factory method creates.

ConcreteProduct: Implements Product interface

Creator: Declares the Factory method

ConcreateCreator: Implements the Factory method to return an instance of a ConcreteProduct

Problem statement: Create a Factory of Games by using Factory Methods, which defines the game interface.

Code snippet:

```
import java.util.HashMap;


/* Product interface as per UML diagram */
interface Game{
    /* createGame is a complex method, which executes a sequence of game steps */
    public void createGame();
}

/* ConcreteProduct implementation as per UML diagram */
class Chess implements Game{
    public Chess(){
        createGame();
    }
    public void createGame(){
        System.out.println("--------------------------------------");
        System.out.println("Create Chess game");
        System.out.println("Opponents:2");
        System.out.println("Define 64 blocks");
        System.out.println("Place 16 pieces for White opponent");
        System.out.println("Place 16 pieces for Black opponent");
        System.out.println("Start Chess game");
        System.out.println("--------------------------------------");
    }
}
class Checkers implements Game{
    public Checkers(){
        createGame();
    }
    public void createGame(){
        System.out.println("--------------------------------------");
        System.out.println("Create Checkers game");
        System.out.println("Opponents:2 or 3 or 4 or 6");
```

```
        System.out.println("For each opponent, place 10 coins");
        System.out.println("Start Checkers game");
        System.out.println("----------------------------------------");
    }
}
class Ludo implements Game{
    public Ludo(){
        createGame();
    }
    public void createGame(){
        System.out.println("----------------------------------------");
        System.out.println("Create Ludo game");
        System.out.println("Opponents:2 or 3 or 4");
        System.out.println("For each opponent, place 4 coins");
        System.out.println("Create two dices with numbers from 1-6");
        System.out.println("Start Ludo game");
        System.out.println("----------------------------------------");
    }
}

/* Creator interface as per UML diagram */
interface IGameFactory {
    public Game getGame(String gameName);
}

/* ConcreteCreator implementation as per UML diagram */
class GameFactory implements IGameFactory {

    HashMap<String,Game> games = new HashMap<String,Game>();
    /*
        Since Game Creation is complex process, we don't want to create game using new
operator every time.
        Instead we create Game only once and store it in Factory. When client request a
specific game,
        Game object is returned from Factory instead of creating new Game on the fly, which is
time consuming
    */

    public GameFactory(){

        games.put(Chess.class.getName(),new Chess());
        games.put(Checkers.class.getName(),new Checkers());
        games.put(Ludo.class.getName(),new Ludo());
    }
    public Game getGame(String gameName){
        return games.get(gameName);
    }
}

public class NonStaticFactoryDemo{
    public static void main(String args[]){
        if ( args.length < 1){
            System.out.println("Usage: java FactoryDemo gameName");
            return;
        }

        GameFactory factory = new GameFactory();
        Game game = factory.getGame(args[0]);
        System.out.println("Game="+game.getClass().getName());
    }
}
```

output:

```
java NonStaticFactoryDemo Chess
--------------------------------------
Create Chess game
Opponents:2
Define 64 blocks
Place 16 pieces for White opponent
Place 16 pieces for Black opponent
Start Chess game
--------------------------------------
--------------------------------------
Create Checkers game
Opponents:2 or 3 or 4 or 6
For each opponent, place 10 coins
Start Checkers game
--------------------------------------
--------------------------------------
Create Ludo game
Opponents:2 or 3 or 4
For each opponent, place 4 coins
Create two dices with numbers from 1-6
Start Ludo game
--------------------------------------
Game=Chess
```

This example shows a `Factory` class by implementing a `FactoryMethod`.

1. `Game` is the interface for all type of games. It defines complex method: `createGame()`
2. `Chess, Ludo, Checkers` are different variants of games, which provide implementation to `createGame()`
3. `public Game getGame(String gameName)` is `FactoryMethod` in `IGameFactory` class
4. `GameFactory` pre-creates different type of games in constructor. It implements `IGameFactory` factory method.
5. game Name is passed as command line argument to `NotStaticFactoryDemo`
6. `getGame` in `GameFactory` accepts a game name and returns corresponding `Game` object.

When to use:

1. ***Factory***: When you don't want to expose object instantiation logic to the client/caller
2. ***Abstract Factory***: When you want to provide interface to families of related or dependent objects without specifying their concrete classes
3. ***Factory Method:*** To define an interface for creating an object, but let the sub-classes decide which class to instantiate

Comparison with other creational patterns:

1. Design 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

2. **Abstract Factory** classes are often implemented with **Factory Methods**, but they can also be implemented using **Prototype**

References for further reading: Sourcemaking design-patterns

Flyweight Factory (C#)

In simple words:

A Flyweight factory that for a given, already known, key will always give the same object as response. For new keys will create the instance and return it.

Using the factory:

```
ISomeFactory<string, object> factory = new FlyweightFactory<string, object>();

var result1 = factory.GetSomeItem("string 1");
var result2 = factory.GetSomeItem("string 2");
var result3 = factory.GetSomeItem("string 1");

//Objects from different keys
bool shouldBeFalse = result1.Equals(result2);

//Objects from same key
bool shouldBeTrue = result1.Equals(result3);
```

Implementation:

```
public interface ISomeFactory<TKey,TResult> where TResult : new()
{
    TResult GetSomeItem(TKey key);
}

public class FlyweightFactory<TKey, TResult> : ISomeFactory<TKey, TResult> where TResult :
new()
{
    public TResult GetSomeItem(TKey key)
    {
        TResult result;
        if(!Mapping.TryGetValue(key, out result))
        {
            result = new TResult();
            Mapping.Add(key, result);
        }
        return result;
    }

    public Dictionary<TKey, TResult> Mapping { get; set; } = new Dictionary<TKey, TResult>();
}
```

Extra Notes

I would recommend to add to this solution the use of an `IoC Container` (as explained in a different example here) instead of creating your own new instances. One can do it by adding a new registration for the `TResult` to the container and then resolving from it (instead of the `dictionary` in

the example).

Factory method

The Factory method pattern is a creational pattern that abstracts away the instantiation logic of an object in order to decouple the client code from it.

When a factory method belongs to a class that is an implementation of another factory pattern such as Abstract factory then it is usually more appropriate to reference the pattern implemented by that class rather than the Factory method pattern.

The Factory method pattern is more commonly referenced when describing a factory method that belongs to a class which is not primarily a factory.

For instance, it may be advantageous to place a factory method on an object that represents a domain concept if that object encapsulates some state that would simplify the creation process of another object. A factory method may also lead to a design that is more aligned with the Ubiquitous Language of a specific context.

Here's a code example:

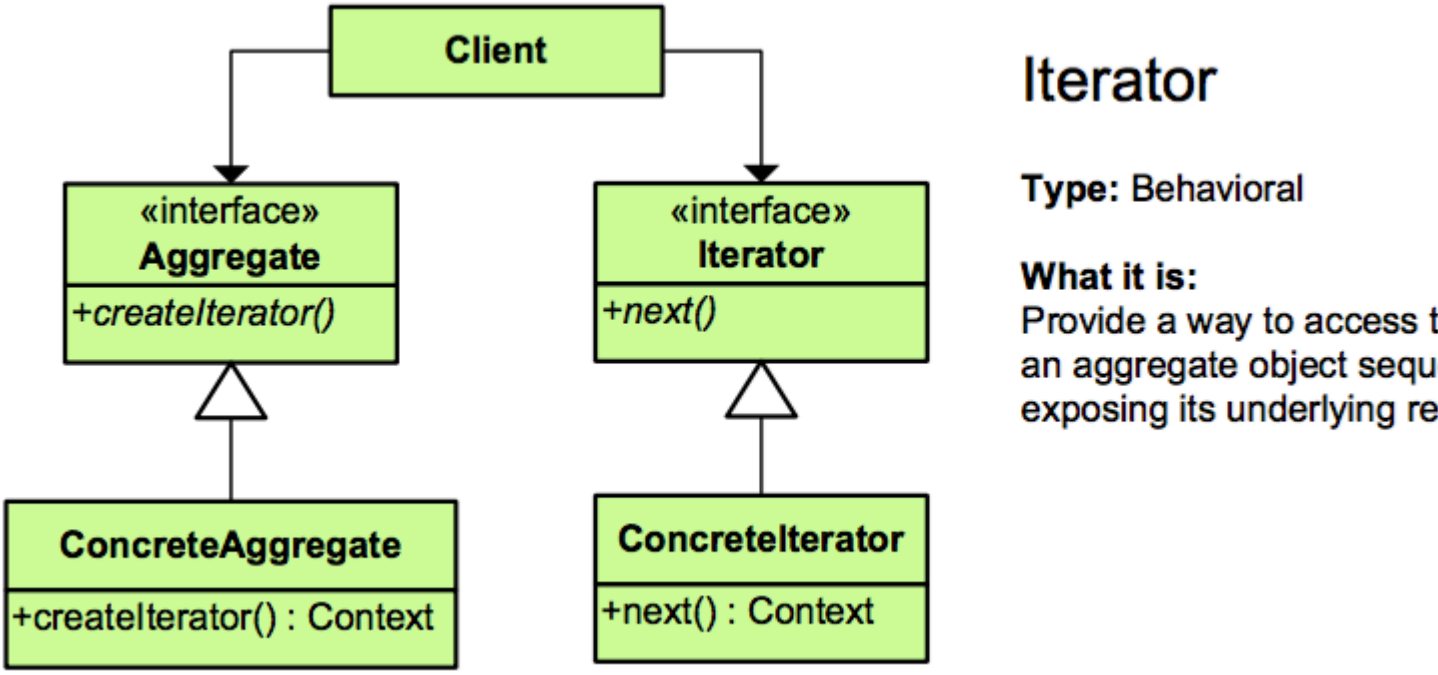
```
//Without a factory method
Comment comment = new Comment(authorId, postId, "This is a comment");

//With a factory method
Comment comment = post.comment(authorId, "This is a comment");
```

Read Factory online: https://riptutorial.com/design-patterns/topic/1375/factory

Chapter 15: Iterator Pattern

Examples

The Iterator Pattern



Collections are one of the most commonly used data structures in software engineering. A Collection is just a group of objects. A collection can be a List, an array, a map, a tree or anything. So, a collection should provide some way to access its elements without exposing its internal structure. We should be able to traverse it in a same irrespective of type of collection it is.

The iterator pattern idea is to take the responsibility of accessing the object of a collection and put it in an iterator object. The iterator object in return will maintain the order of iteration, keep a track of current item and must be having a way to fetch the next element.

Usually, the collection class carries two components: the class itself, and it's `Iterator`.

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}

public class FruitsList {
    public String fruits[] = {"Banana", "Apple", "Pear", "Peach", "Blueberry"};

    public Iterator getIterator() {
       return new FruitIterator();
    }

    private class FruitIterator implements Iterator {
```

```
        int index;

        @Override
        public boolean hasNext() {
            return index < fruits.length;
        }

        @Override
        public Object next() {

           if(this.hasNext()) {
             return names[index++];
           }
           return null;
        }
    }
}
```

Read Iterator Pattern online: https://riptutorial.com/design-patterns/topic/7061/iterator-pattern

Chapter 16: lazy loading

Introduction

eager loading is expensive or the object to be loaded might not be needed at all

Examples

JAVA lazy loading

Call from main()

```
// Simple lazy loader - not thread safe
HolderNaive holderNaive = new HolderNaive();
Heavy heavy = holderNaive.getHeavy();
```

Heavy.class

```
/**
 *
 * Heavy objects are expensive to create.
 *
 */
public class Heavy {

  private static final Logger LOGGER = LoggerFactory.getLogger(Heavy.class);

  /**
   * Constructor
   */
  public Heavy() {
    LOGGER.info("Creating Heavy ...");
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      LOGGER.error("Exception caught.", e);
    }
    LOGGER.info("... Heavy created");
  }
}
```

HolderNaive.class

```
/**
 *
 * Simple implementation of the lazy loading idiom. However, this is not thread safe.
 *
 */
public class HolderNaive {

  private static final Logger LOGGER = LoggerFactory.getLogger(HolderNaive.class);
```

```
  private Heavy heavy;

  /**
   * Constructor
   */
  public HolderNaive() {
    LOGGER.info("HolderNaive created");
  }

  /**
   * Get heavy object
   */
  public Heavy getHeavy() {
    if (heavy == null) {
      heavy = new Heavy();
    }
    return heavy;
  }
}
```

Read lazy loading online: https://riptutorial.com/design-patterns/topic/9951/lazy-loading

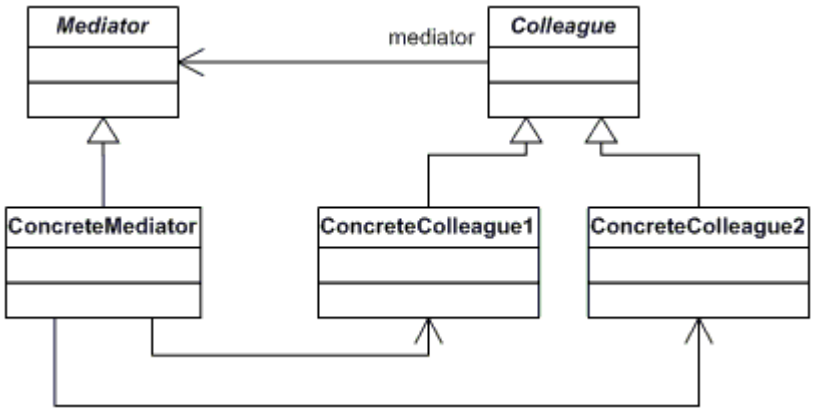
Chapter 17: Mediator Pattern

Examples

Mediator pattern example in java

Mediator pattern defines an object (Mediator) that encapsulates how a set of objects interact. It enables many-to-many communication.

UML diagram:



Key components:

`Mediator`: Defines an interface for communication between Colleagues.

`Colleague`: Is an abstract class, which defines the events to be communicated between Colleagues

`ConcreteMediator`: Implements cooperative behavior by coordinating `Colleague` objects and maintains its colleagues

`ConcreteColleague`: Implements the notification operations received through `Mediator`, which has been generated by other `Colleague`

One real world example:

You are maintaining a network of computers in `Mesh` topology.

A mesh network is a network topology in which each node relays data for the network. All mesh nodes cooperate in the distribution of data in the network.

If a new computer is added Or existing computer is removed, all other computers in that network should know about these two events.

Let's see how Mediator pattern fits into it.

Code snippet:

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

/* Define the contract for communication between Colleagues.
   Implementation is left to ConcreteMediator */
interface Mediator{
    void register(Colleague colleague);
    void unregister(Colleague colleague);
}
/* Define the contract for notification events from Mediator.
   Implementation is left to ConcreteColleague
*/
abstract class Colleague{
    private Mediator mediator;
    private String name;

    public Colleague(Mediator mediator,String name){
        this.mediator = mediator;
        this.name = name;
    }
    public String toString(){
        return name;
    }
    public abstract void receiveRegisterNotification(Colleague colleague);
    public abstract void receiveUnRegisterNotification(Colleague colleague);
}
/*  Process notification event raised by other Colleague through Mediator.
*/
class ComputerColleague extends Colleague {
    private Mediator mediator;

    public ComputerColleague(Mediator mediator,String name){
        super(mediator,name);
    }
    public  void receiveRegisterNotification(Colleague colleague){
        System.out.println("New Computer register event with name:"+colleague+
        ": received @"+this);
        // Send further messages to this new Colleague from now onwards
    }
    public  void receiveUnRegisterNotification(Colleague colleague){
        System.out.println("Computer left unregister event with name:"+colleague+
        ":received @"+this);
        // Do not send further messages to this Colleague from now onwards
    }
}
/* Act as a central hub for communication between different Colleagues.
   Notifies all Concrete Colleagues on occurrence of an event
*/
class NetworkMediator implements Mediator{
    List<Colleague> colleagues = new ArrayList<Colleague>();

    public NetworkMediator(){

    }

    public void register(Colleague colleague){
        colleagues.add(colleague);
        for (Colleague other : colleagues){
            if ( other != colleague){
                other.receiveRegisterNotification(colleague);
            }
```

```
        }
    }
    public void unregister(Colleague colleague){
        colleagues.remove(colleague);
        for (Colleague other : colleagues){
            other.receiveUnRegisterNotification(colleague);
        }
    }
}

public class MediatorPatternDemo{
    public static void main(String args[]){
        Mediator mediator = new NetworkMediator();
        ComputerColleague colleague1 = new ComputerColleague(mediator,"Eagle");
        ComputerColleague colleague2 = new ComputerColleague(mediator,"Ostrich");
        ComputerColleague colleague3 = new ComputerColleague(mediator,"Penguin");
        mediator.register(colleague1);
        mediator.register(colleague2);
        mediator.register(colleague3);
        mediator.unregister(colleague1);
    }
}
```

output:

```
New Computer register event with name:Ostrich: received @Eagle
New Computer register event with name:Penguin: received @Eagle
New Computer register event with name:Penguin: received @Ostrich
Computer left unregister event with name:Eagle:received @Ostrich
Computer left unregister event with name:Eagle:received @Penguin
```

Explanation:

1. `Eagle` is added to network at first through register event. No notifications to any other colleagues since Eagle is the first one.
2. When `Ostrich` is added to the network, `Eagle` is notified : Line 1 of output is rendered now.
3. When `Penguin` is added to network, both `Eagle` and `Ostrich` have been notified : Line 2 and Line 3 of output is rendered now.
4. When `Eagle` left the network through unregister event, both `Ostrich` and `Penguin` have been notified. Line 4 and Line 5 of output is rendered now.

Read Mediator Pattern online: https://riptutorial.com/design-patterns/topic/6184/mediator-pattern

Chapter 18: Monostate

Remarks

As a side note, a few advantages of the `Monostate` pattern over the `Singleton`:

- There is no 'instance` method to be able to access an instance of the class.
- A `Singleton` does not conform to the Java beans notation, but a `Monostate` does.
- Lifetime of instances can be controlled.
- Users of the `Monostate` don't know that they are using a `Monostate`.
- Polymorphism is possible.

Examples

The Monostate Pattern

The `Monostate` pattern is usually referred to as *syntactic sugar* over the `Singleton` pattern or as a *conceptual* *`Singleton`*.

It avoids all the complications of having a single instance of a class, but all the instances use the same data.
This is accomplished mostly by using `static` data members.
One of the most important feature is that it's absolutely transparent for the users, that are completely unaware they are working with a `Monostate`. Users can create as many instances of a `Monostate` as they want and any instance is good as another to access the data.

The `Monostate` class comes usually with a companion class that is used to update the settings if needed.

It follows a minimal example of a `Monostate` in C++:

```
struct Settings {
    Settings() {
        if(!initialized) {
            initialized = true;
            // load from file or db or whatever
            // otherwise, use the SettingsEditor to initialize settings
            Settings::width_ = 42;
            Settings::height_ = 128;
        }
    }

    std::size_t width() const noexcept { return width_; }
    std::size_t height() const noexcept { return height_; }

private:
    friend class SettingsEditor;

    static bool initialized;
```

```
    static std::size_t width_;
    static std::size_t height_;
};

bool Settings::initialized = false;
std::size_t Settings::width_;
std::size_t Settings::height_;

struct SettingsEditor {
    void width(std::size_t value) noexcept { Settings::width_ = value; }
    void height(std::size_t value) noexcept { Settings::height_ = value; }
};
```

Here is an example of a simple implementation of a `Monostate` in Java:

```
public class Monostate {
    private static int width;
    private static int height;

    public int getWidth() {
        return Monostate.width;
    }

    public int getHeight() {
        return Monostate.height;
    }

    public void setWidth(int value) {
        Monostate.width = value;
    }

    public void setHeight(int value) {
        Monostate.height = value;
    }

    static {
        width = 42;
        height = 128;
    }
}
```

Monostate-based hierarchies

In contrasto to the `Singleton`, the `Monostate` is suitable to be inherited to extend its functionalities, as long as member methods are not `static`.
It follows a minimal example in C++:

```
struct Settings {
    virtual std::size_t width() const noexcept { return width_; }
    virtual std::size_t height() const noexcept { return height_; }

private:
    static std::size_t width_;
    static std::size_t height_;
};

std::size_t Settings::width_{0};
```

```
std::size_t Settings::height_{0};

struct EnlargedSettings: Settings {
    std::size_t width() const noexcept override { return Settings::height() + 1; }
    std::size_t height() const noexcept override { return Settings::width()  + 1; }
};
```

Read Monostate online: https://riptutorial.com/design-patterns/topic/6186/monostate

Chapter 19: Multiton

Remarks

Multitonitis

Same as Singleton, Multiton can be considered a bad practice. However, there are times when you can use it wisely (for example, if you building a system like ORM/ODM to persist multiple objects).

Examples

Pool of Singletons (PHP example)

Multiton can be used as a container for singletons. This is Multiton implementation is a combination of Singleton and Pool patterns.

This is an example of how common Multiton abstract Pool class can be created:

```
abstract class MultitonPoolAbstract
{
    /**
     * @var array
     */
    protected static $instances = [];

    final protected function __construct() {}

    /**
     * Get class name of lately binded class
     *
     * @return string
     */
    final protected static function getClassName()
    {
        return get_called_class();
    }

    /**
     * Instantiates a calling class object
     *
     * @return static
     */
    public static function getInstance()
    {
        $className = static::getClassName();

        if( !isset(self::$instances[$className]) ) {
            self::$instances[$className] = new $className;
        }

        return self::$instances[$className];
    }
```

```
    /**
     * Deletes a calling class object
     *
     * @return void
     */
    public static function deleteInstance()
    {
        $className = static::getClassName();

        if( isset(self::$instances[$className]) )
            unset(self::$instances[$className]);
    }

    /*------------------------------------------------------------------------
    | Seal methods that can instantiate the class
    |-------------------------------------------------------------------------*/

    final protected function __clone() {}

    final protected function __sleep() {}

    final protected function __wakeup() {}
}
```

This way we can instantiate a various Singleton pools.

Registry of Singletons (PHP example)

This pattern can be used to contain a registered Pools of Singletons, each distinguished by unique ID:

```
abstract class MultitonRegistryAbstract
{
    /**
     * @var array
     */
    protected static $instances = [];

    /**
     * @param string $id
     */
    final protected function __construct($id) {}

    /**
     * Get class name of lately binded class
     *
     * @return string
     */
    final protected static function getClassName()
    {
        return get_called_class();
    }

    /**
     * Instantiates a calling class object
     *
     * @return static
```

```
     */
    public static function getInstance($id)
    {
        $className = static::getClassName();

        if( !isset(self::$instances[$className]) ) {
            self::$instances[$className] = [$id => new $className($id)];
        } else {
            if( !isset(self::$instances[$className][$id]) ) {
                self::$instances[$className][$id] = new $className($id);
            }
        }

        return self::$instances[$className][$id];
    }

    /**
     * Deletes a calling class object
     *
     * @return void
     */
    public static function unsetInstance($id)
    {
        $className = static::getClassName();

        if( isset(self::$instances[$className]) ) {
            if( isset(self::$instances[$className][$id]) ) {
                unset(self::$instances[$className][$id]);
            }

            if( empty(self::$instances[$className]) ) {
                unset(self::$instances[$className]);
            }
        }
    }

    /*---------------------------------------------------------------------
    | Seal methods that can instantiate the class
    |---------------------------------------------------------------------*/

    final protected function __clone() {}

    final protected function __sleep() {}

    final protected function __wakeup() {}
}
```

This is simplified form of pattern that can be used for ORM to store several entities of a given type.

Read Multiton online: https://riptutorial.com/design-patterns/topic/6857/multiton

Chapter 20: MVC, MVVM, MVP

Remarks

It can be argued that MVC and related patterns are actually software architecture patterns rather than software design patterns.

Examples

Model View Controller (MVC)

1. What is MVC?

The Model View Controller (MVC) Pattern is a design pattern most commonly used for creating user interfaces. The major advantage of MVC is that it separates:

- the internal representation of the application state (the Model),
- how the information is presented to the user (the View), and
- the logic which controls how the user interacts with the application state (the Controller).

2. Use cases of MVC

The primary use case for MVC is in Graphical User Interface (GUI) programming. The View component listens to the Model component for changes. The Model acts as a broadcaster; when there is a change mode to the Model, it broadcasts its changes to the View and the Controller. The Controller is used by the View to modify the Model Component.

3. Implementation

Consider the following implementation of MVC, where we have a Model class called `Animals`, a View class called `DisplayAnimals`, and a controller class called `AnimalController`. The example below is a modified version of the tutorial on MVC from Design Patterns - MVC Pattern.

```
/* Model class */
public class Animals {
    private String name;
    private String gender;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public void setName(String name) {
         this.name = name;
    }
```

```
    public void setGender(String gender) {
        this.gender = gender;
    }
}

/* View class */
public class DisplayAnimals {
    public void printAnimals(String tag, String gender) {
        System.out.println("My Tag name for Animal:" + tag);
        System.out.println("My gender: " + gender);
    }
}

/* Controller class */
public class AnimalController {
     private Animal model;
     private DisplayAnimals view;

   public AnimalController(Animal model, DisplayAnimals view) {
      this.model = model;
      this.view = view;
   }

   public void setAnimalName(String name) {
      model.setName(name);
   }

   public String getAnimalName() {
      return model.getName();
   }

   public void setAnimalGender(String animalGender) {
      model.setGender(animalGender);
   }

   public String getGender() {
      return model.getGender();
   }

   public void updateView() {
      view.printAnimals(model.getName(), model.getGender());
   }
}
```

4. Sources used:

Design Patterns - MVC Pattern

Java SE Application Design With MVC

Model–view–controller

Model View ViewModel (MVVM)

1. What is MVVM?

The Model View ViewModel (MVVM) pattern is a design pattern most commonly used for creating

user interfaces. It is derived from the the popular "Model View Controller" (MVC) pattern. The major advantage of MVVM is that it separates:

- The internal representation of the application state (the Model).
- How the information is presented to the user (the View).
- The "value converter logic" responsible for exposing and converting the data from the model so that the data can be easily managed and presented in the view (the ViewModel).

2. Use cases of MVVM

The primary use case of MVVM is Graphical User Interface (GUI) programming. It is used to simply event-driven programming of user interfaces by separating the view layer from the backend logic managing the data.

In Windows Presentation Foundation (WPF), for example, the view is designed using the framework markup language XAML. The XAML files are bound to ViewModels using data binding. This way the view is only responsible for presentation and the viewmodel is only responsible for managing application state by working on the data in the model.

It is also used in the JavaScript library KnockoutJS.

3. Implementation

Consider the following implementation of MVVM using C# .Net and WPF. We have a Model class called Animals, a View class implemented in Xaml and a ViewModel called AnimalViewModel. The example below is a modified version of the tutorial on MVC from Design Patterns - MVC Pattern.

Look how the Model does not know about anything, the ViewModel only knows about the Model and the View only knows about the ViewModel.

The OnNotifyPropertyChanged-event enables updating both the model and the view so that when you enter something in the textbox in the view the model is updated. And if something updates the model, the view is updated.

```
/*Model class*/
public class Animal
{
    public string Name { get; set; }

    public string Gender { get; set; }
}

/*ViewModel class*/
public class AnimalViewModel : INotifyPropertyChanged
{
    private Animal _model;

    public AnimalViewModel()
    {
        _model = new Animal {Name = "Cat", Gender = "Male"};
    }

    public string AnimalName
```

```
    {
        get { return _model.Name; }
        set
        {
            _model.Name = value;
            OnPropertyChanged("AnimalName");
        }
    }

    public string AnimalGender
    {
        get { return _model.Gender; }
        set
        {
            _model.Gender = value;
            OnPropertyChanged("AnimalGender");
        }
    }

    //Event binds view to ViewModel.
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (this.PropertyChanged != null)
        {
            var e = new PropertyChangedEventArgs(propertyName);
            this.PropertyChanged(this, e);
        }
    }
}


<!-- Xaml View -->
<Window x:Class="WpfApplication6.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"
        xmlns:viewModel="clr-namespace:WpfApplication6">

    <Window.DataContext>
        <viewModel:AnimalViewModel/>
    </Window.DataContext>

    <StackPanel>
        <TextBox Text="{Binding AnimalName}" Width="120" />
        <TextBox Text="{Binding AnimalGender}" Width="120" />
    </StackPanel>
</Window>
```

4. Sources used:

Model–view–viewmodel

A Simple MVVM Example

The World's Simplest C# WPF MVVM Example

The MVVM Pattern

Read MVC, MVVM, MVP online: https://riptutorial.com/design-patterns/topic/7405/mvc--mvvm--mvp

Chapter 21: Null Object pattern

Remarks

Null Object is an object with no referenced value or with defined neutral behaviour. Its purpose is to remove the need of null pointer/reference check.

Examples

Null Object Pattern (C++)

Assuming a abstract class:

```
class ILogger {
    virtual ~ILogger() = default;
    virtual Log(const std::string&) = 0;
};
```

Instead of

```
void doJob(ILogger* logger) {
    if (logger) {
         logger->Log("[doJob]:Step 1");
    }
    // ...
    if (logger) {
         logger->Log("[doJob]:Step 2");
    }
    // ...
    if (logger) {
         logger->Log("[doJob]:End");
    }
}

void doJobWithoutLogging()
{
    doJob(nullptr);
}
```

You may create a Null Object Logger:

```
class NullLogger : public ILogger
{
    void Log(const std::string&) override { /* Empty */ }
};
```

and then change `doJob` in the following:

```
void doJob(ILogger& logger) {
    logger.Log("[doJob]:Step1");
```

```
    // ...
    logger.Log("[doJob]:Step 2");
    // ...
    logger.Log("[doJob]:End");
}

void doJobWithoutLogging()
{
    NullLogger logger;
    doJob(logger);
}
```

Null Object Java using enum

Given an interface:

```
public interface Logger {
   void log(String message);
}
```

Rather than usage:

```
public void doJob(Logger logger) {
    if (logger != null) {
       logger.log("[doJob]:Step 1");
    }
    // ...
    if (logger != null) {
       logger.log("[doJob]:Step 2");
    }
    // ...
    if (logger != null) {
       logger.log("[doJob]:Step 3");
    }
}

public void doJob() {
    doJob(null); // Without Logging
}
```

Because null objects have no state, it makes sense to use a enum singleton for it, so given a null object implemented like so:

```
public enum NullLogger implements Logger {
    INSTANCE;

    @Override
    public void log(String message) {
        // Do nothing
    }
}
```

You can then avoid the null checks.

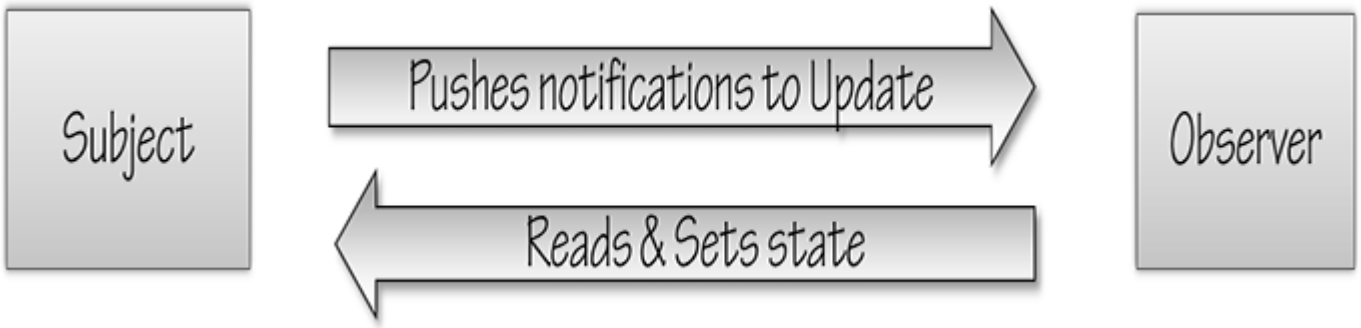
```
public void doJob(Logger logger) {
```

```
    logger.log("[doJob]:Step 1");
    // ...
    logger.log("[doJob]:Step 2");
    // ...
    logger.log("[doJob]:Step 3");
}

public void doJob() {
    doJob(NullLogger.INSTANCE);
}
```

Read Null Object pattern online: https://riptutorial.com/design-patterns/topic/6177/null-object-pattern

Chapter 22: Observer

Remarks



What is the intent ?

- Adopt the principle of separation of concerns.
- Create a separation between the subject and the observer.
- Allow multiple observers to react to change a single subject.

What is the structure ?

- Subject provides a way to Register, Unregister, Notify.
- Observer provides a way to update.

Examples

Observer / Java

The observer pattern lets users of a class subscribe to events that happen when this class processes data etc. and be notified when these event occur. In the following example we create a processing class and an observer class which will be notified while processing a phrase - if it finds words that are longer than 5 letters.

The `LongWordsObserver` interface defines the observer. Implement this interface in order to register an observer to events.

```
// an observe that can be registered and receive notifications
public interface LongWordsObserver {
    void notify(WordEvent event);
}
```

The `WordEvent` class is the event that will be sent to the observer classes once certain events occur (in this case, long words were found)

```
// An event class which contains the long word that was found
public class WordEvent {
```

```
    private String word;

    public WordEvent(String word) {
        this.word = word;
    }

    public String getWord() {
        return word;
    }
}
```

The `PhraseProcessor` class is the class that processes the given phrase. It allows observers to be registered using the `addObserver` method. Once long words are found, these observers will be called using an instance of the `WordEvent` class.

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

public class PhraseProcessor {

    // the list of observers
    private List<LongWordsObserver> observers = new ArrayList<>();

    // register an observer
    public void addObserver(LongWordsObserver observer) {
        observers.add(observer);
    }

    // inform all the observers that a long word was found
    private void informObservers(String word) {
        observers.forEach(o -> o.notify(new WordEvent(word)));
    }

    // the main method - process a phrase and look for long words. If such are found,
    // notify all the observers
    public void process(String phrase) {
        for (String word : phrase.split(" ")) {
            if (word.length() > 5) {
                informObservers(word);
            }
        }
    }
}
```

The `LongWordsExample` class shows how to register observers, call the `process` method and receive alerts when long words were found.

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

public class LongWordsExample {

    public static void main(String[] args) {

        // create a list of words to be filled when long words were found
        List<String> longWords = new ArrayList<>();
```

```
        // create the PhraseProcessor class
        PhraseProcessor processor = new PhraseProcessor();

        // register an observer and specify what it should do when it receives events,
        // namely to append long words in the longwords list
        processor.addObserver(event -> longWords.add(event.getWord()));

        // call the process method
        processor.process("Lorem ipsum dolor sit amet, consectetuer adipiscing elit");

        // show the list of long words after the processing is done
        System.out.println(String.join(", ", longWords));
        // consectetuer, adipiscing
    }
}
```

Observer using IObservable and IObserver (C#)

`IObserver<T>` and `IObservable<T>` interfaces can be used to implement observer pattern in .NET

- `IObservable<T>` interface represents the class that sends notifications
- `IObserver<T>` interface represents the class that receives them

```
public class Stock {
  private string Symbol { get; set; }
  private decimal Price { get; set; }
}

public class Investor : IObserver<Stock> {
  public IDisposable unsubscriber;
  public virtual void Subscribe(IObservable<Stock> provider) {
    if(provider != null) {
      unsubscriber = provider.Subscribe(this);
    }
  }
  public virtual void OnCompleted() {
    unsubscriber.Dispose();
  }
  public virtual void OnError(Exception e) {
  }
  public virtual void OnNext(Stock stock) {
  }
}

public class StockTrader : IObservable<Stock> {
  public StockTrader() {
    observers = new List<IObserver<Stock>>();
  }
  private IList<IObserver<Stock>> observers;
  public IDisposable Subscribe(IObserver<Stock> observer) {
    if(!observers.Contains(observer)) {
      observers.Add(observer);
    }
    return new Unsubscriber(observers, observer);
  }
  public class Unsubscriber : IDisposable {
    private IList<IObserver<Stock>> _observers;
    private IObserver<Stock> _observer;
```

```
    public Unsubscriber(IList<IObserver<Stock>> observers, IObserver<Stock> observer) {
      _observers = observers;
      _observer = observer;
    }

    public void Dispose() {
      Dispose(true);
    }
    private bool _disposed = false;
    protected virtual void Dispose(bool disposing) {
      if(_disposed) {
        return;
      }
      if(disposing) {
        if(_observer != null && _observers.Contains(_observer)) {
          _observers.Remove(_observer);
        }
      }
      _disposed = true;
    }
  }
  public void Trade(Stock stock) {
    foreach(var observer in observers) {
      if(stock== null) {
        observer.OnError(new ArgumentNullException());
      }
      observer.OnNext(stock);
    }
  }
  public void End() {
    foreach(var observer in observers.ToArray()) {
      observer.OnCompleted();
    }
    observers.Clear();
  }
}
```

Usage

```
...
var provider = new StockTrader();
var i1 = new Investor();
i1.Subscribe(provider);
var i2 = new Investor();
i2.Subscribe(provider);

provider.Trade(new Stock());
provider.Trade(new Stock());
provider.Trade(null);
provider.End();
...
```

REF: Design patterns and practices in .NET: the Observer pattern

Read Observer online: https://riptutorial.com/design-patterns/topic/3185/observer

Chapter 23: Open Close Principle

Introduction

The Open Close Principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged. Software entities like classes, modules and functions should be open for extension but closed for modifications.

Remarks

Like every principle Open Close Principle is only a principle. Making a flexible design involves additional time and effort spent for it and it introduce new level of abstraction increasing the complexity of the code. So this principle should be applied in those area which are most likely to be changed. There are many design patterns that help us to extend code without changing it, for example decorator.

Examples

Open Close Principle violation

```
/*
* This design have some major issues
* For each new shape added the unit testing
* of the GraphicEditor should be redone
* When a new type of shape is added the time
* for adding it will be high since the developer
* who add it should understand the logic
* of the GraphicEditor.
* Adding a new shape might affect the existing
* functionality in an undesired way,
* even if the new shape works perfectly
*/

class GraphicEditor {
    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r) {....}
    public void drawRectangle(Rectangle r) {....}
}

 class Shape {
     int m_type;
 }
```

```
class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}

class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}
```

Open Close Principle support

```
/*
* For each new shape added the unit testing
* of the GraphicEditor should not be redone
* No need to understand the sourcecode
* from GraphicEditor.
* Since the drawing code is moved to the
* concrete shape classes, it's a reduced risk
* to affect old functionallity when new
* functionallity is added.
*/
class GraphicEditor {
     public void drawShape(Shape s) {
         s.draw();
     }
 }

 class Shape {
     abstract void draw();
 }

 class Rectangle extends Shape  {
     public void draw() {
         // draw the rectangle
     }
 }
```

Read Open Close Principle online: https://riptutorial.com/design-patterns/topic/9199/open-close-principle

Chapter 24: Prototype Pattern

Introduction

The Prototype pattern is a creational pattern that creates new objects by cloning existing prototype object. The prototype pattern speeds up the instantiation of classes when copying objects is faster.

Remarks

The prototype pattern is a creational design pattern. It 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 when a class need a *"polymorphic (copy) constructor"*.

Examples

Prototype Pattern (C++)

```
class IPrototype  {
public:
    virtual ~IPrototype() = default;

    auto Clone() const { return std::unique_ptr<IPrototype>{DoClone()}; }
    auto Create() const { return std::unique_ptr<IPrototype>{DoCreate()}; }

private:
    virtual IPrototype* DoClone() const = 0;
    virtual IPrototype* DoCreate() const = 0;
};

class A : public IPrototype {
public:
    auto Clone() const { return std::unique_ptr<A>{DoClone()}; }
    auto Create() const { return std::unique_ptr<A>{DoCreate()}; }
private:
    // Use covariant return type :)
    A* DoClone() const override { return new A(*this); }
    A* DoCreate() const override { return new A; }
};

class B : public IPrototype {
public:
    auto Clone() const { return std::unique_ptr<B>{DoClone()}; }
    auto Create() const { return std::unique_ptr<B>{DoCreate()}; }
private:
    // Use covariant return type :)
    B* DoClone() const override { return new B(*this); }
    B* DoCreate() const override { return new B; }
};


class ChildA : public A {
```

```
public:
    auto Clone() const { return std::unique_ptr<ChildA>{DoClone()}; }
    auto Create() const { return std::unique_ptr<ChildA>{DoCreate()}; }

private:
    // Use covariant return type :)
    ChildA* DoClone() const override { return new ChildA(*this); }
    ChildA* DoCreate() const override { return new ChildA; }
};
```

That allows to construct the derived class from a base class pointer:

```
ChildA childA;
A& a = childA;
IPrototype& prototype = a;

// Each of the following will create a copy of `ChildA`:
std::unique_ptr<ChildA> clone1 = childA.Clone();
std::unique_ptr<A> clone2 = a.Clone();
std::unique_ptr<IPrototype> clone3 = prototype.Clone();

// Each of the following will create a new default instance `ChildA`:
std::unique_ptr<ChildA> instance1 = childA.Create();
std::unique_ptr<A> instance2 = a.Create();
std::unique_ptr<IPrototype> instance3 = prototype.Create();
```

Prototype Pattern (C#)

The prototype pattern can be implemented using the ICloneable interface in .NET.

```
class Spoon {
}
class DessertSpoon : Spoon, ICloneable {
  ...
  public object Clone() {
    return this.MemberwiseClone();
  }
}
class SoupSpoon : Spoon, ICloneable {
  ...
  public object Clone() {
    return this.MemberwiseClone();
  }
}
```

Prototype Pattern (JavaScript)

In the classical languages like Java, C# or C++ we start by creating a class and then we can create new objects from the class or we can extend the class.

In JavaScript first we create an object, then we can augment the object or create new objects from it. So i think, JavaScript demonstrates actual prototype than the classical language.

Example :

```
var myApp = myApp || {};

myApp.Customer = function (){
    this.create = function () {
        return "customer added";
    }
};

myApp.Customer.prototype = {
    read: function (id) {
        return "this is the customer with id = " + id;
    },
    update: function () {
        return "customer updated";
    },
    remove: function () {
        return "customer removed";
    }
};
```

Here, we create an object named `Customer`, and then without creating *new object* we extended the existing `Customer object` using *prototype* keyword. This technique is known as **Prototype Pattern**.

Read Prototype Pattern online: https://riptutorial.com/design-patterns/topic/5867/prototype-pattern

Chapter 25: Publish-Subscribe

Examples

Publish-Subscribe in Java

The publisher-subscriber is a familiar concept given the rise of YouTube, Facebook and other social media services. The basic concept is that there is a `Publisher` who generates content and a `Subscriber` who consumes content. Whenever the `Publisher` generates content, each `Subscriber` is notified. `Subscribers` can theoretically be subscribed to more than one publisher.

Usually there is a `ContentServer` that sits between publisher and subscriber to help mediate the messaging

```
public class Publisher {
    ...
    public Publisher(Topic t) {
        this.topic = t;
    }

    public void publish(Message m) {
        ContentServer.getInstance().sendMessage(this.topic, m);
    }
}
```

```
public class ContentServer {
    private Hashtable<Topic, List<Subscriber>> subscriberLists;

    private static ContentServer serverInstance;

    public static ContentServer getInstance() {
        if (serverInstance == null) {
            serverInstance = new ContentServer();
        }
        return serverInstance;
    }

    private ContentServer() {
        this.subscriberLists = new Hashtable<>();
    }

    public sendMessage(Topic t, Message m) {
        List<Subscriber> subs = subscriberLists.get(t);
        for (Subscriber s : subs) {
            s.receivedMessage(t, m);
        }
    }

    public void registerSubscriber(Subscriber s, Topic t) {
        subscriberLists.get(t).add(s);
    }
```

```
public class Subscriber {
```

```
    public Subscriber(Topic...topics) {
        for (Topic t : topics) {
            ContentServer.getInstance().registerSubscriber(this, t);
        }
    }

    public void receivedMessage(Topic t, Message m) {
        switch(t) {
            ...
        }
    }
}
```

Usually, the pub-sub design pattern is implemented with a multithreaded view in mind. One of the more common implementations sees each `Subscriber` as a separate thread, with the `ContentServer` managing a thread pool

Simple pub-sub example in JavaScript

Publishers and subscribers don't need to know each other. They simply communicate with the help of message queues.

```
(function () {
        var data;

        setTimeout(function () {
            data = 10;
            $(document).trigger("myCustomEvent");
        }, 2000);

        $(document).on("myCustomEvent", function () {
            console.log(data);
        });
})();
```

Here we published a custom event named **myCustomEvent** and subscribed on that event. So they don't need to know each other.

Read Publish-Subscribe online: https://riptutorial.com/design-patterns/topic/7260/publish-subscribe

Chapter 26: Repository Pattern

Remarks

About the implementation of `IEnumerable<TEntity> Get(Expression<Func<TEntity, bool>> filter)`: The idea of this is to use Expressions like `i => x.id == 17` to write generic requests. It is a way to query data without using the specific query language of your technology. The implementation is rather extensive, therefore you might want to consider other alternatives, like specific methods on your implemented repositories: An imaginary `CompanyRepository` could provide the method `GetByName(string name)`.

Examples

Read-only repositories (C#)

A repository pattern can be used to encapsulate data storage specific code in designated components. The part of your application, that needs the data will only work with the repositories. You will want to create a repository for each combination of item you store and your database technology.

Read only repositories can be used to create repositories that are not allowed to manipulate data.

The interfaces

```
public interface IReadOnlyRepository<TEntity, TKey> : IRepository
{
    IEnumerable<TEntity> Get(Expression<Func<TEntity, bool>> filter);

    TEntity Get(TKey id);
}

public interface IRepository<TEntity, TKey> : IReadOnlyRepository<TEntity, TKey>
{
    TKey Add(TEntity entity);

    bool Delete(TKey id);

    TEntity Update(TKey id, TEntity entity);
}
```

An example implementation using ElasticSearch as technology (with NEST)

```
public abstract class ElasticReadRepository<TModel> : IReadOnlyRepository<TModel, string>
```

```
    where TModel : class
{

    protected ElasticClient Client;

    public ElasticReadRepository()
    {
        Client = Connect();
    }

    protected abstract ElasticClient Connect();

    public TModel Get(string id)
    {
        return Client.Get<TModel>(id).Source;
    }

    public IEnumerable<TModel> Get(Expression<Func<TModel, bool>> filter)
    {
        /* To much code for this example */
        throw new NotImplementedException();
    }
}

public abstract class ElasticRepository<TModel>
    : ElasticReadRepository<TModel>, IRepository<TModel, string>
    where TModel : class
{
    public string Add(TModel entity)
    {
        return Client.Index(entity).Id;
    }

    public bool Delete(string id)
    {
        return Client.Delete<TModel>(id).Found;
    }

    public TModel Update(string id, TModel entity)
    {
        return Connector.Client.Update<TModel>(
            id,
            update => update.Doc(entity)
        ).Get.Source;
    }
}
```

Using this implementation, you can now create specific Repositories for the items you want to store or access. When using elastic search, it is common, that some components should only read the data, thus read-only repositories should be used.

Repository Pattern using Entity Framework (C#)

Repository interface;

```
public interface IRepository<T>
{
    void Insert(T entity);
```

```
    void Insert(ICollection<T> entities);
    void Delete(T entity);
    void Delete(ICollection<T> entity);
    IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate);
    IQueryable<T> GetAll();
    T GetById(int id);
}
```

Generic repository;

```
public class Repository<T> : IRepository<T> where T : class
{
    protected DbSet<T> DbSet;

    public Repository(DbContext dataContext)
    {
        DbSet = dataContext.Set<T>();
    }

    public void Insert(T entity)
    {
        DbSet.Add(entity);
    }

    public void Insert(ICollection<T> entities)
    {
        DbSet.AddRange(entities);
    }

    public void Delete(T entity)
    {
        DbSet.Remove(entity);
    }

    public void Delete(ICollection<T> entities)
    {
        DbSet.RemoveRange(entities);
    }

    public IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate)
    {
        return DbSet.Where(predicate);
    }

    public IQueryable<T> GetAll()
    {
        return DbSet;
    }

    public T GetById(int id)
    {
        return DbSet.Find(id);
    }
}
```

Example use using a demo hotel class;

```
var db = new DatabaseContext();
var hotelRepo = new Repository<Hotel>(db);
```

```
var hotel = new Hotel("Hotel 1", "42 Wallaby Way, Sydney");
hotelRepo.Insert(hotel);
db.SaveChanges();
```

Read Repository Pattern online: https://riptutorial.com/design-patterns/topic/6254/repository-pattern

Chapter 27: Singleton

Remarks

The Singleton design pattern is sometimes regarded as "*Anti pattern*". This is due to the fact that it has some problems. You have to decide for yourself if you think it is appropriate to use it. This topic has been discussed several times on StackOverflow.

See: http://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons

Examples

Singleton (C#)

Singletons are used to ensure that only one instance of an object is being created. The singleton allows only a single instance of itself to be created which means it controls its creation. The singleton is one of the Gang of Four design patterns and is a **creational pattern**.

Thread-Safe Singleton Pattern

```
public sealed class Singleton
{
    private static Singleton _instance;
    private static object _lock = new object();

    private Singleton()
    {
    }

    public static Singleton GetSingleton()
    {
        if (_instance == null)
        {
             CreateSingleton();
        }

        return _instance;
    }

    private static void CreateSingleton()
    {
        lock (_lock )
        {
            if (_instance == null)
            {
                 _instance = new Singleton();
            }
        }
    }
}
```

Jon Skeet provides the following implementation for a lazy, thread-safe singleton:

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton());

    public static Singleton Instance { get { return lazy.Value; } }

    private Singleton()
    {
    }
}
```

Singleton (Java)

Singletons in Java are very similar to C#, being as both languages are object orientated. Below is an example of a singleton class, where only one version of the object can be alive during the program's lifetime (Assuming the program works on one thread)

```
public class SingletonExample {

    private SingletonExample() { }

    private static SingletonExample _instance;

    public static SingletonExample getInstance() {

        if (_instance == null) {
            _instance = new SingletonExample();
        }
        return _instance;
    }
}
```

Here is the thread safe version of that program:

```
public class SingletonThreadSafeExample {

    private SingletonThreadSafeExample () { }

    private static volatile SingletonThreadSafeExample _instance;

    public static SingletonThreadSafeExample getInstance() {
        if (_instance == null) {
                createInstance();
        }
        return _instance;
    }

    private static void createInstance() {
        synchronized(SingletonThreadSafeExample.class) {
            if (_instance == null) {
                _instance = new SingletonThreadSafeExample();
            }
        }
```

```
    }
}
```

Java also have an object called `ThreadLocal`, which creates a single instance of an object on a thread by thread basis. This could be useful in applications where each thread need its own version of the object

```
public class SingletonThreadLocalExample {

    private SingletonThreadLocalExample () { }

    private static ThreadLocal<SingletonThreadLocalExample> _instance = new
ThreadLocal<SingletonThreadLocalExample>();

    public static SingletonThreadLocalExample getInstance() {
        if (_instance.get() == null) {
            _instance.set(new SingletonThreadLocalExample());
        }
        return _instance.get();
    }
}
```

Here is also a **Singleton** implementation with `enum` (containing only one element):

```
public enum SingletonEnum {
    INSTANCE;
    // fields, methods
}
```

Any **Enum** class implementation ensures that there is *only one* instance of its each element will exist.

Bill Pugh Singleton Pattern

Bill Pugh Singleton Pattern is the most widely used approach for Singleton class as it doesn’t require synchronization

```
public class SingletonExample {

    private SingletonExample(){}

    private static class SingletonHolder{
        private static final SingletonExample INSTANCE = new SingletonExample();
    }

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

with using private inner static class the holder is not loaded to memory until someone call the getInstance method. Bill Pugh solution is thread safe and it doesn't required synchronization.

There are more Java singleton examples in the Singletons topic under the Java documentation tag.

Singleton (C++)

As per Wiki : In software engineering, the singleton pattern is a design pattern that restricts the instantiation of a class to one object.

This is required to create exactly one object to coordinate actions across the system.

```
class Singleton
{
    // Private constructor so it can not be arbitrarily created.
    Singleton()
    {}
    // Disable the copy and move
    Singleton(Singleton const&)            = delete;
    Singleton& operator=(Singleton const&) = delete;
  public:

    // Get the only instance
    static Singleton& instance()
    {
        // Use static member.
        // Lazily created on first call to instance in thread safe way (after C++ 11)
        // Guaranteed to be correctly destroyed on normal application exit.
        static Singleton _instance;

        // Return a reference to the static member.
        return _instance;
    }
};
```

Lazy Singleton practical example in java

Real life use cases for Singleton pattern;

If you are developing a client-server application, you need single instrance of `ConnectionManager`, which manages the life cycle of client connections.

The basic APIs in ConnectionManager :

`registerConnection`: Add new connection to existing list of connections

`closeConnection` : Close the connection either from event triggered by Client or Server

`broadcastMessage` : Some times, you have to send a message to all subscribed client connections.

I am not providing complete implementation of source code since example will become very lengthy. At high level, the code will be like this.

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

```
/* Lazy Singleton - Thread Safe Singleton without synchronization and volatile constructs */
final class  LazyConnectionManager {
    private Map<String,Connection> connections = new HashMap<String,Connection>();
    private LazyConnectionManager() {}
    public static LazyConnectionManager getInstance() {
        return LazyHolder.INSTANCE;
    }
    private static class LazyHolder {
        private static final LazyConnectionManager INSTANCE = new LazyConnectionManager();
    }

    /* Make sure that De-Serailzation does not create a new instance */
    private Object readResolve()  {
        return LazyHolder.INSTANCE;
    }
    public void registerConnection(Connection connection){
        /* Add new connection to list of existing connection */
        connections.put(connection.getConnectionId(),connection);
    }
    public void closeConnection(String connectionId){
        /* Close connection and remove from map */
        Connection connection = connections.get(connectionId);
        if ( connection != null) {
            connection.close();
            connections.remove(connectionId);
        }
    }
    public void broadcastMessage(String message){
        for (Map.Entry<String, Connection> entry : connections.entrySet()){
            entry.getValue().sendMessage(message);
        }
    }
}
```

Sample Server class:

```
class Server implements Runnable{
    ServerSocket socket;
    int id;
    public Server(){
        new Thread(this).start();
    }
    public void run(){
        try{
            ServerSocket socket = new ServerSocket(4567);
            while(true){
                Socket clientSocket = socket.accept();
                ++id;
                Connection connection = new Connection(""+ id,clientSocket);
                LazyConnectionManager.getInstance().registerConnection(connection);
                LazyConnectionManager.getInstance().broadcastMessage("Message pushed by
server:");
            }
        }catch(Exception err){
            err.printStackTrace();
        }
    }

}
```

Other practical use cases for Singletons:

1. Managing global resources like `ThreadPool, ObjectPool, DatabaseConnectionPool` etc.
2. Centralised services like `Logging` application data with different log levels like `DEBUG,INFO,WARN,ERROR` etc
3. Global `RegistryService` where different services are registered with a central component on startup.That global service can act as a `Facade` for the application

C# Example: Multithreaded Singleton

Static initialization is suitable for most situations. When your application must delay the instantiation, use a non-default constructor or perform other tasks before the instantiation, and work in a multithreaded environment, you need a different solution. Cases do exist, however, in which you cannot rely on the common language runtime to ensure thread safety, as in the Static Initialization example. In such cases, you must use specific language capabilities to ensure that only one instance of the object is created in the presence of multiple threads. One of the more common solutions is to use the Double-Check Locking [Lea99] idiom to keep separate threads from creating new instances of the singleton at the same time.

The following implementation allows only a single thread to enter the critical area, which the lock block identifies, when no instance of Singleton has yet been created:

```
using System;

public sealed class Singleton {
   private static volatile Singleton instance;
   private static object syncRoot = new Object();

   private Singleton() {}

   public static Singleton Instance    {
      get
      {
         if (instance == null)
         {
            lock (syncRoot)
            {
               if (instance == null)
                  instance = new Singleton();
            }
         }

         return instance;
      }
  }
}
```

This approach ensures that only one instance is created and only when the instance is needed. Also, the variable is declared to be volatile to ensure that assignment to the instance variable completes before the instance variable can be accessed. Lastly, this approach uses a syncRoot instance to lock on, rather than locking on the type itself, to avoid deadlocks.

This double-check locking approach solves the thread concurrency problems while avoiding an

exclusive lock in every call to the Instance property method. It also allows you to delay instantiation until the object is first accessed. In practice, an application rarely requires this type of implementation. In most cases, the static initialization approach is sufficient.

Reference: MSDN

Acknowledgments

[Gamma95] Gamma, Helm, Johnson, and Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Lea99] Lea, Doug. Concurrent Programming in Java, Second Edition. Addison-Wesley, 1999.

[Sells03] Sells, Chris. "Sealed Sucks." sellsbrothers.com News. Available at: http://www.sellsbrothers.com/news/showTopic.aspx?ixTopic=411.

Singleton (PHP)

Example from phptherightway.com

```
<?php
class Singleton
{
    /**
     * @var Singleton The reference to *Singleton* instance of this class
     */
    private static $instance;

    /**
     * Returns the *Singleton* instance of this class.
     *
     * @return Singleton The *Singleton* instance.
     */
    public static function getInstance()
    {
        if (null === static::$instance) {
            static::$instance = new static();
        }

        return static::$instance;
    }

    /**
     * Protected constructor to prevent creating a new instance of the
     * *Singleton* via the `new` operator from outside of this class.
     */
    protected function __construct()
    {
    }

    /**
     * Private clone method to prevent cloning of the instance of the
     * *Singleton* instance.
     *
     * @return void
     */
```

```
    private function __clone()
    {
    }

    /**
     * Private unserialize method to prevent unserializing of the *Singleton*
     * instance.
     *
     * @return void
     */
    private function __wakeup()
    {
    }
}

class SingletonChild extends Singleton
{
}

$obj = Singleton::getInstance();
var_dump($obj === Singleton::getInstance());             // bool(true)

$anotherObj = SingletonChild::getInstance();
var_dump($anotherObj === Singleton::getInstance());      // bool(false)

var_dump($anotherObj === SingletonChild::getInstance()); // bool(true)
```

Singleton Design pattern (in general)

Note: The singleton is a design pattern.
But it also considered an anti-pattern.

The use of a singleton should be considered carefully before use. There are usually better alternatives.

The main problem with a singleton is the same as the problem with global variables. They introduce external global mutable state. This means that functions that use a singleton are not solely dependent on the input parameters but also the state of the singleton. This means that testing can be severely compromised (difficult).

The issues with singletons can be mitigated by using them in conjunction with creation patterns; so that the initial creation of the singleton can be controlled.

Read Singleton online: https://riptutorial.com/design-patterns/topic/2179/singleton

Chapter 28: SOLID

Introduction

What is S.O.L.I.D ?

S.O.L.I.D. is a mnemonic(memory aid) acronym. The Solid principles should help software developers to avoid „code smells“ and should lead to good sourcecode. Good sourcecode means in this context that the sourcecode is easy to extend and maintain. The main focus of the Solid principles are classes

What to expect:

Why you should apply S.O.L.I.D

How to apply the five S.O.L.I.D principles (examples)

Examples

SRP - Single Responsibility Principle

The S in S.O.L.I.D stands for Single responsibility principle(SRP).

Responsibility means in this context reasons to change, so the principle states that a class should only have one reason to change.

Robert C. Martin stated it (during his lecture at Yale shool of management in 10 Sep 2014) as follows

> You could also say, don't put functions that change for different reasons in the same class.

or

> Don't mix concerns in your classes

Reason to apply the SRP:

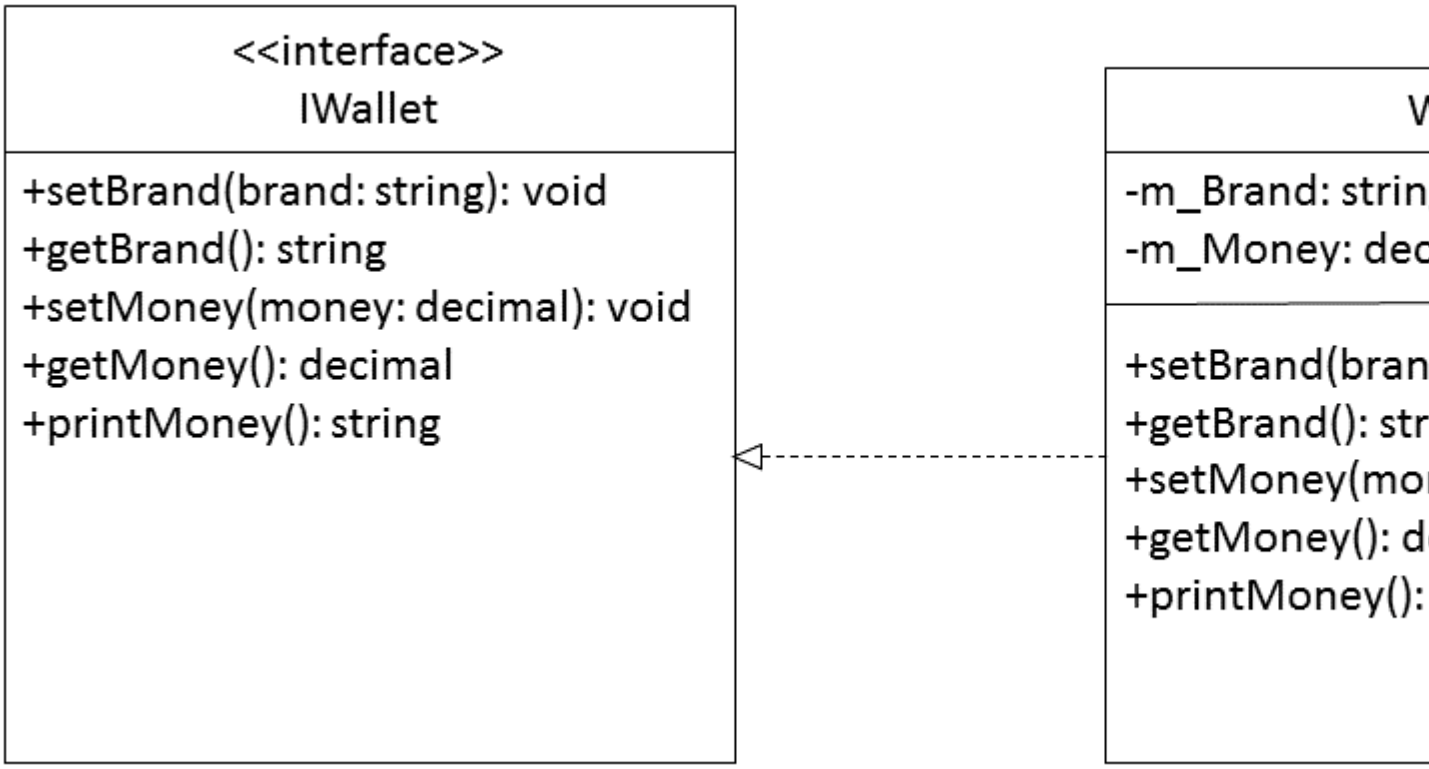
When you change a class you might affect functionality related to other responsibilities of the class. Keeping the responsibilities at a low level minimizes the risk of side effects.

Bad example

We have an interface IWallet and a Wallet class which implements the IWallet. The Wallet holds our money and the brand, furthermore should it print our money as string representation. The class is used by

1. a webservice

2. a textwriter which prints the money in Euros into a textfile.



The SRP is here violated because we have two concerns:

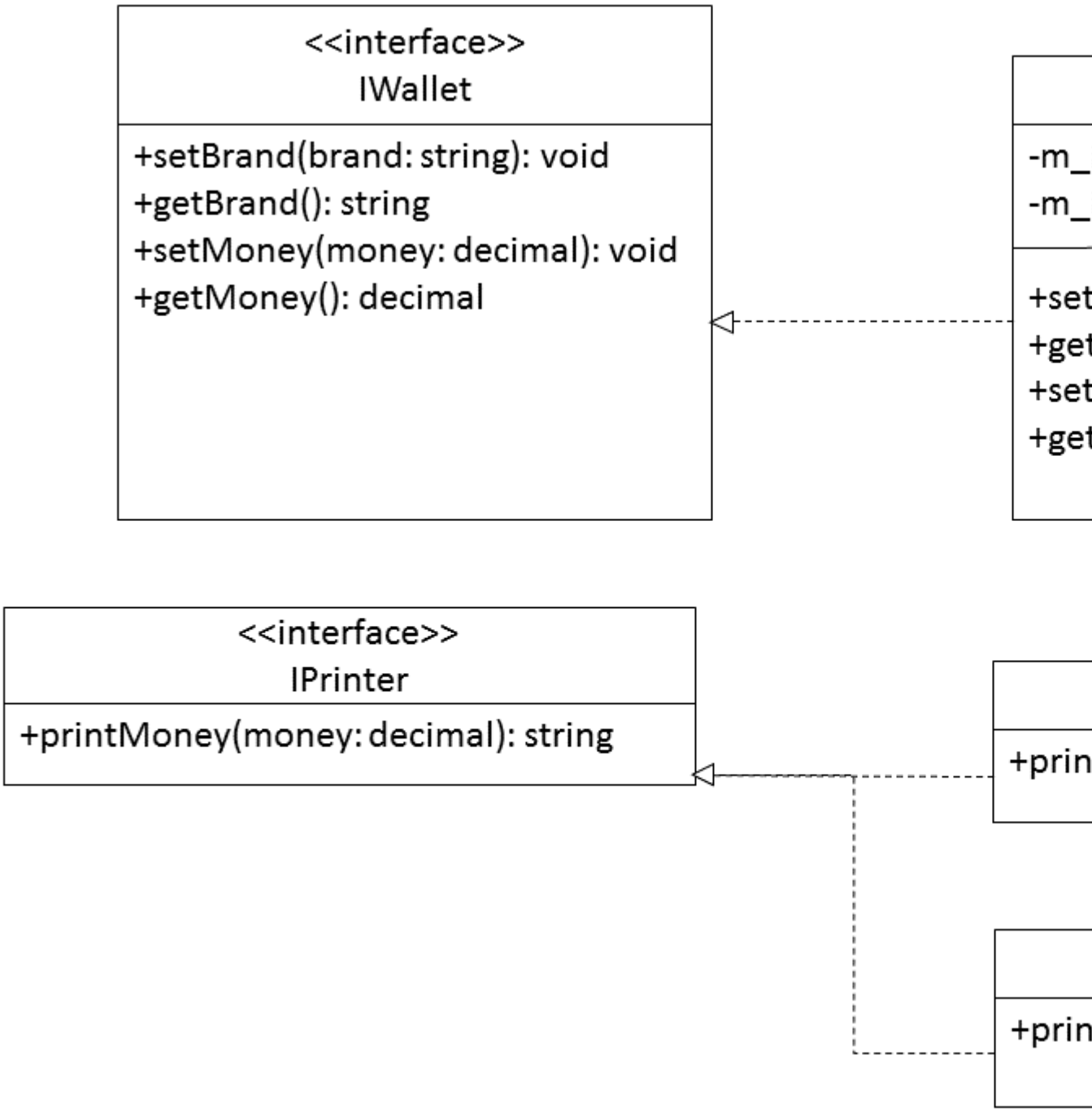
1. The storing of the money and brand
2. The representation of the money.

C# example code

```
public interface IWallet
{
    void setBrand(string brand);
    string getBrand();
    void setMoney(decimal money);
    decimal getMoney();
    string printMoney();
}

public class Wallet : IWallet
{
    private decimal m_Money;
    private string m_Brand;

    public string getBrand()
    {
        return m_Brand;
    }

    public decimal getMoney()
    {
        return m_Money;
    }
```

```
    public void setBrand(string brand)
    {
        m_Brand = brand;
    }

    public void setMoney(decimal money)
    {
        m_Money = money;
    }

    public string printMoney()
    {
        return m_Money.ToString();
    }
}
```

Good example



To avoid the violation of the SRP, we removed the `printMoney` method from the Wallet class and placed it into a Printer class. The Printer class is now responsible for the printing and the Wallet is now responsible for the storing of the values.

C# example code

```
public interface IPrinter
{
    void printMoney(decimal money);
}

public class EuroPrinter : IPrinter
{
    public void printMoney(decimal money)
```

```
    {
        //print euro
    }
}

public class DollarPrinter : IPrinter
{
    public void printMoney(decimal money)
    {
        //print Dollar
    }
}

public interface IWallet
{
    void setBrand(string brand);
    string getBrand();
    void setMoney(decimal money);
    decimal getMoney();
}

public class Wallet : IWallet
{
    private decimal m_Money;
    private string m_Brand;

    public string getBrand()
    {
        return m_Brand;
    }

    public decimal getMoney()
    {
        return m_Money;
    }

    public void setBrand(string brand)
    {
        m_Brand = brand;
    }

    public void setMoney(decimal money)
    {
        m_Money = money;
    }
}
```

Read SOLID online: https://riptutorial.com/design-patterns/topic/8651/solid

Chapter 29: Static factory method

Examples

Static Factory method

We can provide a meaningful name for our constructors.

We can provide several constructors with the same number and type of parameters, something that as we saw earlier we can't do with class constructors.

```
public class RandomIntGenerator {
    private final int min;
    private final int max;

    private RandomIntGenerator(int min, int max) {
        this.min = min;
        this.max = max;
    }

    public static RandomIntGenerator between(int max, int min) {
        return new RandomIntGenerator(min, max);
    }

    public static RandomIntGenerator biggerThan(int min) {
        return new RandomIntGenerator(min, Integer.MAX_VALUE);
    }

    public static RandomIntGenerator smallerThan(int max) {
        return new RandomIntGenerator(Integer.MIN_VALUE, max);
    }

    public int next() {...}
}
```

Hiding direct access to constructor

We can avoid providing direct access to resource intensive constructors, like for databases. public class DbConnection { private static final int MAX_CONNS = 100; private static int totalConnections = 0;

```
   private static Set<DbConnection> availableConnections = new HashSet<DbConnection>();

   private DbConnection()
   {
     // ...
     totalConnections++;
   }

   public static DbConnection getDbConnection()
   {
     if(totalConnections < MAX_CONNS)
     {
```

```
      return new DbConnection();
    }

    else if(availableConnections.size() > 0)
    {
        DbConnection dbc = availableConnections.iterator().next();
        availableConnections.remove(dbc);
        return dbc;
    }

    else {
      throw new NoDbConnections();
    }
  }

  public static void returnDbConnection(DbConnection dbc)
  {
    availableConnections.add(dbc);
    //...
  }
}
```

Static Factory Method C#

The *static factory method* is a variation of the *factory method* pattern. It is used to create objects without having to call the constructor yourself.

When to use the Static Factory Method

- if you want to give a meaningfull name to the method that generates your object.
- if you want to avoid overcomplex object creation see Tuple Msdn.
- if you want to limit the number of objects created (caching)
- if you want to return an object of any subtype of their return type.

There are some disadvantages like

- Classes without a public or protected constructor cannot be initialized in the static factory method.
- Static factory methods are like normal static methods, so they are not distinguishable from other static methods (this may vary from IDE to IDE)

Example

Pizza.cs

```
public class Pizza
{
    public int SizeDiameterCM
    {
        get;
        private set;
    }

    private Pizza()
    {
```

```
        SizeDiameterCM = 25;
    }

    public static Pizza GetPizza()
    {
        return new Pizza();
    }

    public static Pizza GetLargePizza()
    {
        return new Pizza()
        {
            SizeDiameterCM = 35
        };
    }

    public static Pizza GetSmallPizza()
    {
        return new Pizza()
        {
            SizeDiameterCM = 28
        };
    }

    public override string ToString()
    {
        return String.Format("A Pizza with a diameter of {0} cm",SizeDiameterCM);
    }
}
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        var pizzaNormal = Pizza.GetPizza();
        var pizzaLarge = Pizza.GetLargePizza();
        var pizzaSmall = Pizza.GetSmallPizza();


        String pizzaString = String.Format("{0} and {1} and {2}",pizzaSmall.ToString(),
pizzaNormal.ToString(), pizzaLarge.ToString());
        Console.WriteLine(pizzaString);
    }
}
```

Output

> A Pizza with a diameter of 28 cm and A Pizza with a diameter of 25 cm and A Pizza with a diameter of 35 cm

Read Static factory method online: https://riptutorial.com/design-patterns/topic/6024/static-factory-method

Chapter 30: Strategy Pattern

Examples

Hiding strategy implementation details

A very common guideline in object oriented design is "as little as possible but as much as necessary". This also applies to the strategy pattern: It is usually advisable to hide implementation details, for example which classes actually implement strategies.

For simple strategies which do not depend on external parameters, the most common approach is to make the implementing class itself private (nested classes) or package-private and exposing an instance through a static field of a public class:

```
public class Animal {

  private static class AgeComparator implements Comparator<Animal> {
    public int compare(Animal a, Animal b) {
      return a.age() - b.age();
    }
  }

  // Note that this field has the generic type Comparator<Animal>, *not*
  // Animal.AgeComparator!
  public static final Comparator<Animal> AGE_COMPARATOR = new AgeComparator();

  private final int age;

  Animal(int age) {
    this.age = age;
  }

  public int age() {
    return age;
  }

}

List<Animal> myList = new LinkedList<>();
myList.add(new Animal(10));
myList.add(new Animal(5));
myList.add(new Animal(7));
myList.add(new Animal(9));

Collections.sort(
  myList,
  Animal.AGE_COMPARATOR
);
```

The public field `Animal.AGE_COMPARATOR` defines a strategy which can then be used in methods such as `Collections.sort`, but it does not require the user to know anything about its implementation, not even the implementing class.

If you prefer, you can use an anonymous class:

```
public class Animal {

  public static final Comparator<Animal> AGE_COMPARATOR = new Comparator<Animal> {
    public int compare(Animal a, Animal b) {
      return a.age() - b.age();
    }
  };

  // other members...
}
```

If the strategy is a little more complex and requires parameters, it is very common to use static factory methods such as `Collections.reverseOrder(Comparator<T>)`. The return type of the method should not expose any implementation details, e.g. `reverseOrder()` is implemented like

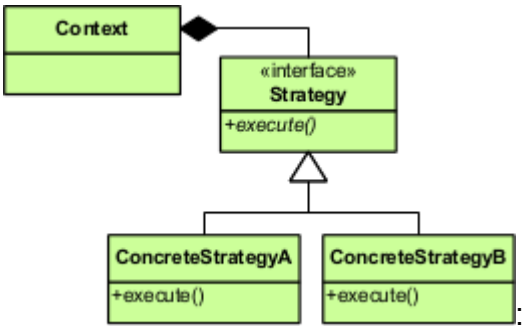
```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {
  // (Irrelevant lines left out.)
  return new ReverseComparator2<>(cmp);
}
```

Strategy pattern example in java with Context class

Strategy:

`Strategy` is a behavioural pattern, which allows to change the algorithm dynamically from a family of related algorithms.

UML of Strategy pattern from Wikipedia




```
import java.util.*;

/* Interface for Strategy */
interface OfferStrategy {
    public String getName();
    public double getDiscountPercentage();
}
/* Concrete implementation of base Strategy */
class NoDiscountStrategy implements OfferStrategy{
    public String getName(){
        return this.getClass().getName();
    }
    public double getDiscountPercentage(){
```

```
        return 0;
    }
}
/* Concrete implementation of base Strategy */
class QuarterDiscountStrategy implements OfferStrategy{
    public String getName(){
        return this.getClass().getName();
    }
    public double getDiscountPercentage(){
        return 0.25;
    }
}
/* Context is optional. But if it is present, it acts as single point of contact
   for client.

   Multiple uses of Context
   1. It can populate data to execute an operation of strategy
   2. It can take independent decision on Strategy creation.
   3. In absence of Context, client should be aware of concrete strategies. Context acts a
wrapper and hides internals
   4. Code re-factoring will become easy
*/
class StrategyContext {
    double price; // price for some item or air ticket etc.
    Map<String,OfferStrategy> strategyContext = new HashMap<String,OfferStrategy>();
    StrategyContext(double price){
        this.price= price;
        strategyContext.put(NoDiscountStrategy.class.getName(),new NoDiscountStrategy());
        strategyContext.put(QuarterDiscountStrategy.class.getName(),new
QuarterDiscountStrategy());
    }
    public void applyStrategy(OfferStrategy strategy){
        /*
        Currently applyStrategy has simple implementation. You can Context for populating some
more information,
        which is required to call a particular operation
        */
        System.out.println("Price before offer :"+price);
        double finalPrice = price - (price*strategy.getDiscountPercentage());
        System.out.println("Price after offer:"+finalPrice);
    }
    public OfferStrategy getStrategy(int monthNo){
        /*
            In absence of this Context method, client has to import relevant concrete
Strategies everywhere.
            Context acts as single point of contact for the Client to get relevant Strategy
        */
        if ( monthNo < 6 )  {
            return strategyContext.get(NoDiscountStrategy.class.getName());
        }else{
            return strategyContext.get(QuarterDiscountStrategy.class.getName());
        }

    }
}
public class StrategyDemo{
    public static void main(String args[]){
        StrategyContext context = new StrategyContext(100);
        System.out.println("Enter month number between 1 and 12");
        int month = Integer.parseInt(args[0]);
        System.out.println("Month ="+month);
```

```
        OfferStrategy strategy = context.getStrategy(month);
        context.applyStrategy(strategy);
    }

}
```

output:

```
Enter month number between 1 and 12
Month =1
Price before offer :100.0
Price after offer:100.0

Enter month number between 1 and 12
Month =7
Price before offer :100.0
Price after offer:75.0
```

Problem statement: Offer 25% discount on price of item for the months of July-December. Do not provide any discount for the months of Jan-June.

Above example shows the usage of `Strategy` pattern with `Context`. `Context` can be used as Single Point of Contact for the `Client`.

Two Strategies - `NoOfferStrategy` and `QuarterDiscountStrategy` have been declared as per problem statement.

As shown in the output column, you will get discount depending on the month you have entered

Use case(s) for Strategy pattern:

1. Use this pattern when you have a family of interchangeable algorithms and you have to change the algorithm at run time.
2. Keep the code cleaner by removing conditional statements

Strategy pattern without a context class / Java

The following is a simple example of using the strategy pattern without a context class. There are two implementation strategies which implement the interface and solve the same problem in different ways. Users of the EnglishTranslation class can call the translate method and choose which strategy they would like to use for the translation, by specifying the desired strategy.

```
// The strategy interface
public interface TranslationStrategy {
    String translate(String phrase);
}

// American strategy implementation
public class AmericanTranslationStrategy implements TranslationStrategy {

    @Override
    public String translate(String phrase) {
```

```
        return phrase + ", bro";
    }
}

// Australian strategy implementation
public class AustralianTranslationStrategy implements TranslationStrategy {

    @Override
    public String translate(String phrase) {
        return phrase + ", mate";
    }
}

// The main class which exposes a translate method
public class EnglishTranslation {

    //  translate a phrase using a given strategy
    public static String translate(String phrase, TranslationStrategy strategy) {
        return strategy.translate(phrase);
    }

    // example usage
    public static void main(String[] args) {

        // translate a phrase using the AustralianTranslationStrategy class
        String aussieHello = translate("Hello", new AustralianTranslationStrategy());
        // Hello, mate

        // translate a phrase using the AmericanTranslationStrategy class
        String usaHello = translate("Hello", new AmericanTranslationStrategy());
        // Hello, bro
    }
}
```

Using Java 8 functional interfaces to implement the Strategy pattern

The purpose of this example is to show how we can realize the Strategy pattern using Java 8 functional interfaces. We will start with a simple use case codes in classic Java, and then recode it in the Java 8 way.

The example problem we using is a family of algorithms (strategies) that *describe* different ways to communicate over a distance.

The Classic Java version

The contract for our family of algorithms is defined by the following interface:

```
public interface CommunicateInterface {
    public String communicate(String destination);
}
```

Then we can implement a number of algorithms, as follows:

```
public class CommunicateViaPhone implements CommunicateInterface {
    @Override
```

```
    public String communicate(String destination) {
        return "communicating " + destination +" via Phone..";
    }
}

public class CommunicateViaEmail implements CommunicateInterface {
    @Override
    public String communicate(String destination) {
        return "communicating " + destination + " via Email..";
    }
}

public class CommunicateViaVideo implements CommunicateInterface {
    @Override
    public String communicate(String destination) {
        return "communicating " + destination + " via Video..";
    }
}
```

These can be instantiated as follows:

```
CommunicateViaPhone communicateViaPhone = new CommunicateViaPhone();
CommunicateViaEmail communicateViaEmail = new CommunicateViaEmail();
CommunicateViaVideo communicateViaVideo = new CommunicateViaVideo();
```

Next, we implement a service that uses the strategy:

```
public class CommunicationService {
    private CommunicateInterface communcationMeans;

    public void setCommuncationMeans(CommunicateInterface communcationMeans) {
        this.communcationMeans = communcationMeans;
    }

    public void communicate(String destination) {
        this.communcationMeans.communicate(destination);
    }
}
```

Finally, we can use the different strategies as follows:

```
CommunicationService communicationService = new CommunicationService();

// via phone
communicationService.setCommuncationMeans(communicateViaPhone);
communicationService.communicate("1234567");

// via email
communicationService.setCommuncationMeans(communicateViaEmail);
communicationService.communicate("hi@me.com");
```

Using Java 8 functional interfaces

The contract of the different algorithm implementations does not need a dedicated interface. Instead, we can describe it using the existing `java.util.function.Function<T, R>` interface.

The different algorithms composing `the family of algorithms` can be expressed as lambda expressions. This replaces the strategy classes *and* their instantiations.

```
Function<String, String> communicateViaEmail =
        destination -> "communicating " + destination + " via Email..";
Function<String, String> communicateViaPhone =
        destination -> "communicating " + destination + " via Phone..";
Function<String, String> communicateViaVideo =
        destination -> "communicating " + destination + " via Video..";
```

Next, we can code the "service" as follows:

```
public class CommunicationService {
    private Function<String, String> communcationMeans;

    public void setCommuncationMeans(Function<String, String> communcationMeans) {
        this.communcationMeans = communcationMeans;
    }

    public void communicate(String destination) {
        this.communcationMeans.communicate(destination);
    }
}
```

Finally we use the strategies as follows

```
CommunicationService communicationService = new CommunicationService();

// via phone
communicationService.setCommuncationMeans(communicateViaPhone);
communicationService.communicate("1234567");

// via email
communicationService.setCommuncationMeans(communicateViaEmail);
communicationService.communicate("hi@me.com");
```

Or even:

```
communicationService.setCommuncationMeans(
    destination -> "communicating " + destination + " via Smoke signals.." );
CommunicationService.communicate("anyone");
```

Strategy (PHP)

Example from www.phptherightway.com

```
<?php

interface OutputInterface
{
    public function load();
}

class SerializedArrayOutput implements OutputInterface
```

```
{
    public function load()
    {
        return serialize($arrayOfData);
    }
}

class JsonStringOutput implements OutputInterface
{
    public function load()
    {
        return json_encode($arrayOfData);
    }
}

class ArrayOutput implements OutputInterface
{
    public function load()
    {
        return $arrayOfData;
    }
}
```

Read Strategy Pattern online: https://riptutorial.com/design-patterns/topic/1331/strategy-pattern

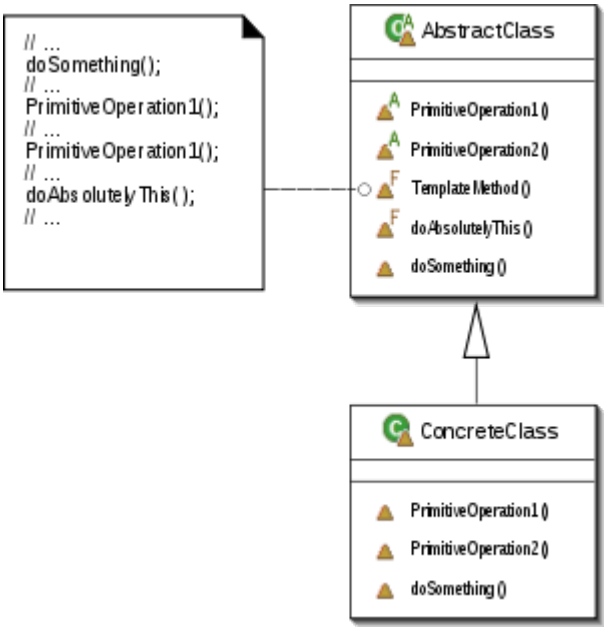
Chapter 31: Template Method

Examples

Template method implementation in java

Template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in an operation, defering some steps to subclasses.

Structure:



Key notes:

1. Template method uses Inheritance
2. The Template method implemented by the base class should not be overridden. In this way, the structure of the algorithm is controlled by the super class, and the details are implemented in the sub classes

Code example:

```
import java.util.List;

class GameRule{

}
class GameInfo{
    String gameName;
    List<String> players;
    List<GameRule> rules;
}

abstract class Game{
    protected GameInfo info;
```

```
    public Game(GameInfo info){
        this.info = info;
    }
    public abstract void createGame();
    public abstract void makeMoves();
    public abstract void applyRules();

    /* playGame is template method. This algorithm skeleton can't be changed by sub-classes.
sub-class can change
       the behaviour only of steps like createGame() etc. */

    public void playGame(){
        createGame();
        makeMoves();
        applyRules();
        closeGame();
    }
    protected void closeGame(){
        System.out.println("Close game:"+this.getClass().getName());
        System.out.println("--------------------");
    }
}
class Chess extends Game{
    public Chess(GameInfo info){
        super(info);
    }
    public void createGame(){
        // Use GameInfo and create Game
        System.out.println("Creating Chess game");
    }
    public void makeMoves(){
        System.out.println("Make Chess moves");
    }
    public void applyRules(){
        System.out.println("Apply Chess rules");
    }
}
class Checkers extends Game{
    public Checkers(GameInfo info){
        super(info);
    }
    public void createGame(){
        // Use GameInfo and create Game
        System.out.println("Creating Checkers game");
    }
    public void makeMoves(){
        System.out.println("Make Checkers moves");
    }
    public void applyRules(){
        System.out.println("Apply Checkers rules");
    }

}
class Ludo extends Game{
    public Ludo(GameInfo info){
        super(info);
    }
    public void createGame(){
        // Use GameInfo and create Game
        System.out.println("Creating Ludo game");
    }
```

```
    public void makeMoves(){
        System.out.println("Make Ludo moves");
    }
    public void applyRules(){
        System.out.println("Apply Ludo rules");
    }
}

public class TemplateMethodPattern{
    public static void main(String args[]){
        System.out.println("--------------------");

        Game game = new Chess(new GameInfo());
        game.playGame();

        game = new Ludo(new GameInfo());
        game.playGame();

        game = new Checkers(new GameInfo());
        game.playGame();
    }
}
```

Explanation:

1. `Game` is an `abstract` super class, which defines a template method : `playGame()`
2. Skeleton of `playGame()` is defined in base class: `Game`
3. Sub-classes like `Chess, Ludo` and `Checkers` can't change the skeleton of `playGame()`. But they can modify the behaviour of some steps like

```
createGame();
makeMoves();
applyRules();
```

output:

```
--------------------
Creating Chess game
Make Chess moves
Apply Chess rules
Close game:Chess
--------------------
Creating Ludo game
Make Ludo moves
Apply Ludo rules
Close game:Ludo
--------------------
Creating Checkers game
Make Checkers moves
Apply Checkers rules
Close game:Checkers
--------------------
```

Read Template Method online: https://riptutorial.com/design-patterns/topic/4867/template-method

Chapter 32: Visitor Pattern

Examples

Visitor Pattern example in C++

Instead of

```
struct IShape
{
    virtual ~IShape() = default;

    virtual void print() const = 0;
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    // .. and so on
};
```

Visitors can be used:

```
// The concrete shapes
struct Square;
struct Circle;

// The visitor interface
struct IShapeVisitor
{
    virtual ~IShapeVisitor() = default;
    virtual void visit(const Square&) = 0;
    virtual void visit(const Circle&) = 0;
};

// The shape interface
struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};
```

Now the concrete shapes:

```
struct Point {
    double x;
    double y;
};

struct Circle : IShape
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}

    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
```

```
    Point center;
    double radius;
};

struct Square : IShape
{
    Square(const Point& topLeft, double sideLength) :
         topLeft(topLeft), sideLength(sideLength)
    {}

    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }

    Point topLeft;
    double sideLength;
};
```

then the visitors:

```
struct ShapePrinter : IShapeVisitor
{
    void visit(const Square&) override { std::cout << "Square"; }
    void visit(const Circle&) override { std::cout << "Circle"; }
};

struct ShapeAreaComputer : IShapeVisitor
{
    void visit(const Square& square) override
    {
        area = square.sideLength * square.sideLength;
    }

    void visit(const Circle& circle) override
    {
         area = M_PI * circle.radius * circle.radius;
    }

    double area = 0;
};

struct ShapePerimeterComputer : IShapeVisitor
{
    void visit(const Square& square) override { perimeter = 4. * square.sideLength; }
    void visit(const Circle& circle) override { perimeter = 2. * M_PI * circle.radius; }

    double perimeter = 0.;
};
```

And use it:

```
const Square square = {{-1., -1.}, 2.};
const Circle circle{{0., 0.}, 1.};
const IShape* shapes[2] = {&square, &circle};

ShapePrinter shapePrinter;
ShapeAreaComputer shapeAreaComputer;
ShapePerimeterComputer shapePerimeterComputer;

for (const auto* shape : shapes) {
```

```
    shape->accept(shapePrinter);
    std::cout << " has an area of ";

    // result will be stored in shapeAreaComputer.area
    shape->accept(shapeAreaComputer);

    // result will be stored in shapePerimeterComputer.perimeter
    shape->accept(shapePerimeterComputer);

    std::cout << shapeAreaComputer.area
              << ", and a perimeter of "
              << shapePerimeterComputer.perimeter
              << std::endl;
}
```

Expected output:

```
Square has an area of 4, and a perimeter of 8
Circle has an area of 3.14159, and a perimeter of 6.28319
```

Demo

Explanation:

- In `void Square::accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }`, the static type of `this` is known, and so the chosen (at compile time) overload is `void IVisitor::visit(const Square&);`.
- For `square.accept(visitor);` call, the dynamic dispatch through `virtual` is used to know which `accept` to call.

Pros:

- You may add new functionality (`SerializeAsXml`, ...) to the class `IShape` just by adding a new visitor.

Cons:

- Adding a new concrete shape (`Triangle`, ...) requires to modifying all visitors.

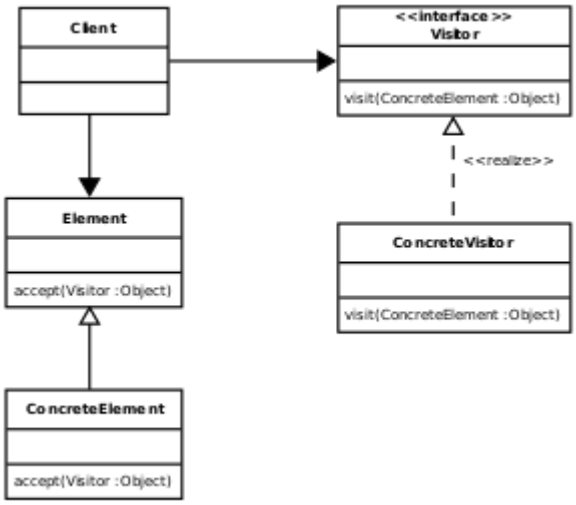
The alternative of putting all functionalities as `virtual` methods in `IShape` has opposite pros and cons: Adding new functionality requires to modify all existing shapes, but adding a new shape doesn't impact existing classes.

Visitor pattern example in java

`Visitor` pattern allows you to add new operations or methods to a set of classes without modifying the structure of those classes.

This pattern is especially useful when you want to centralise a particular operation on an object without extending the object Or without modifying the object.

UML diagram from wikipedia:



Code snippet:

```
import java.util.HashMap;

interface Visitable{
    void accept(Visitor visitor);
}

interface Visitor{
    void logGameStatistics(Chess chess);
    void logGameStatistics(Checkers checkers);
    void logGameStatistics(Ludo ludo);
}
class GameVisitor implements Visitor{
    public void logGameStatistics(Chess chess){
        System.out.println("Logging Chess statistics: Game Completion duration, number of
moves etc..");
    }
    public void logGameStatistics(Checkers checkers){
        System.out.println("Logging Checkers statistics: Game Completion duration, remaining
coins of loser");
    }
    public void logGameStatistics(Ludo ludo){
        System.out.println("Logging Ludo statistics: Game Completion duration, remaining coins
of loser");
    }
}

abstract class Game{
    // Add game related attributes and methods here
    public Game(){

    }
    public void getNextMove(){};
    public void makeNextMove(){}
    public abstract String getName();
}
class Chess extends Game implements Visitable{
    public String getName(){
        return Chess.class.getName();
    }
```

```
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}
class Checkers extends Game implements Visitable{
    public String getName(){
        return Checkers.class.getName();
    }
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}
class Ludo extends Game implements Visitable{
    public String getName(){
        return Ludo.class.getName();
    }
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}

public class VisitorPattern{
    public static void main(String args[]){
        Visitor visitor = new GameVisitor();
        Visitable games[] = { new Chess(),new Checkers(), new Ludo()};
        for (Visitable v : games){
            v.accept(visitor);
        }
    }
}
```

Explanation:

1. `Visitable` (`Element`) is an interface and this interface method has to be added to a set of classes.
2. `Visitor` is an interface, which contains methods to perform an operation on `Visitable` elements.
3. `GameVisitor` is a class, which implements `Visitor` interface (`ConcreteVisitor`).
4. Each `Visitable` element accept `Visitor` and invoke a relevant method of `Visitor` interface.
5. You can treat `Game` as `Element` and concrete games like `Chess,Checkers and Ludo` as `ConcreteElements`.

In above example, `Chess, Checkers and Ludo` are three different games (and `Visitable` classes). On one fine day, I have encountered with a scenario to log statistics of each game. So without modifying individual class to implement statistics functionality, you can centralise that responsibility in `GameVisitor` class, which does the trick for you without modifying the structure of each game.

output:

```
Logging Chess statistics: Game Completion duration, number of moves etc..
Logging Checkers statistics: Game Completion duration, remaining coins of loser
Logging Ludo statistics: Game Completion duration, remaining coins of loser
```

Use cases/Applicability:

1. *Similar operations have to be performed* on objects of different types grouped in a structure
2. You need to execute many distinct and unrelated operations. *It separates Operation from objects Structure*
3. New operations have to be added *without change in object structure*
4. *Gather related operations into a single class* rather than force you to change or derive classes
5. Add functions to class libraries for which you *either do not have the source or cannot change the source*

Additional references:

oodesign

sourcemaking

Visitor Example in C++

```
// A simple class hierarchy that uses the visitor to add functionality.
//
class VehicleVisitor;
class Vehicle
{
    public:
        // To implement the visitor pattern
        // The class simply needs to implement the accept method
        // That takes a reference to a visitor object that provides
        // new functionality.
        virtual void accept(VehicleVisitor& visitor) = 0
};
class Plane: public Vehicle
{
    public:
        // Each concrete representation simply calls the visit()
        // method on the visitor object passing itself as the parameter.
        virtual void accept(VehicleVisitor& visitor) {visitor.visit(*this);}

        void fly(std::string const& destination);
};
class Train: public Vehicle
{
    public:
        virtual void accept(VehicleVisitor& visitor) {visitor.visit(*this);}

        void locomote(std::string const& destination);
};
class Automobile: public Vehicle
{
    public:
        virtual void accept(VehicleVisitor& visitor) {visitor.visit(*this);}

        void drive(std::string const& destination);
};

class VehicleVisitor
{
    public:
```

```
        // The visitor interface implements one method for each class in the
        // hierarchy. When implementing new functionality you just create the
        // functionality required for each type in the appropriate method.

        virtual void visit(Plane& object)      = 0;
        virtual void visit(Train& object)      = 0;
        virtual void visit(Automobile& object) = 0;

    // Note: because each class in the hierarchy needs a virtual method
    // in visitor base class this makes extending the hierarchy ones defined
    // hard.
};
```

An example usage:

```
// Add the functionality `Move` to an object via a visitor.
class MoveVehicleVisitor
{
    std::string const& destination;
    public:
        MoveVehicleVisitor(std::string const& destination)
            : destination(destination)
    {}
    virtual void visit(Plane& object)      {object.fly(destination);}
    virtual void visit(Train& object)      {object.locomote(destination);}
    virtual void visit(Automobile& object) {object.drive(destination);}
};

int main()
{
    MoveVehicleVisitor  moveToDenver("Denver");
    Vehicle&            object = getObjectToMove();
    object.accept(moveToDenver);
}
```

Traversing large objects

The visitor Pattern can be used to traverse structures.

```
class GraphVisitor;
class Graph
{
    public:
        class Node
        {
            using Link = std::set<Node>::iterator;
            std::set<Link>   linkTo;
            public:
                void accept(GraphVisitor& visitor);
        };

        void accept(GraphVisitor& visitor);

    private:
        std::set<Node>  nodes;
};

class GraphVisitor
```

```
{
    std::set<Graph::Node*>  visited;
    public:
        void visit(Graph& graph)
        {
            visited.clear();
            doVisit(graph);
        }
        bool visit(Graph::Node& node)
        {
            if (visited.find(&node) != visited.end()) {
                return false;
            }
            visited.insert(&node);
            doVisit(node);
            return true;
        }
    private:
        virtual void doVisit(Graph& graph)      = 0;
        virtual void doVisit(Graph::Node& node) = 0;
};

void accept(GraphVisitor& visitor)
{
    // Pass the graph to the visitor.
    visitor.visit(*this);

    // Then do a depth first search of the graph.
    // In this situation it is the visitors responsibility
    // to keep track of visited nodes.
    for(auto& node: nodes) {
        node.accept(visitor);
    }
}
void Graph::Node::accept(GraphVisitor& visitor)
{
    // Tell the visitor it is working on a node and see if it was
    // previously visited.
    if (visitor.visit(*this)) {

        // The pass the visitor to all the linked nodes.
        for(auto& link: linkTo) {
            link->accept(visitor);
        }
    }
}
```

Read Visitor Pattern online: https://riptutorial.com/design-patterns/topic/4579/visitor-pattern

Credits

S. No	Chapters	Contributors
1	Getting started with Design patterns	Community, Ekin, Mateen Ulhaq, meJustAndrew, Sahan Serasinghe, Saurabh Sarode, Stephen C, Tim, ɬolɐəz əɥɬ qoq
2	Adapter	avojak, Ben Rhys-Lewis, Daniel Käfer, deHaar, Thijs Riezebeek
3	blackboard	Leonidas Menendez
4	Bridge Pattern	Mark, Ravindra babu, Vasiliy Vlasov
5	Builder Pattern	Alexey Groshev, Arif, Daniel Käfer, fgb, Kyle Morgan, Ravindra babu, Sujit Kamthe, uzilan, Vasiliy Vlasov, yitzih
6	Chain of Responsibility	Ben Rhys-Lewis
7	Command pattern	matiaslauriti, Ravindra babu, Vasiliy Vlasov
8	Composite pattern	Krzysztof Branicki
9	Data Access Object(DAO) design pattern	Pritam Banerjee
10	Decorator pattern	Arif, Krzysztof Branicki, matiaslauriti, Ravindra babu
11	Dependency Injection	Kritner, matiaslauriti, user2321864
12	Facade	Kritner, Makoto, Ravindra babu
13	Factory	Adil Abbasi, Daniel Käfer, Denis Elkhov, FireAlkazar, Geeky Ninja, Gilad Green, Jarod42, Jean-Baptiste Yunès, kstandell, Leifb, matiaslauriti, Michael Brown, Nijin22, plalx, Ravindra babu , Stephen Byrne, Tejas Pawar
14	Iterator Pattern	bw_üezi, Dave Ranjan, Jeeter, Stephen C
15	lazy loading	Adhikari Bishwash
16	Mediator Pattern	Ravindra babu, Vasiliy Vlasov
17	Monostate	skypjack

18	Multiton	Kid Binary
19	MVC, MVVM, MVP	Daniel LIn, Jompa234, Stephen C, user1223339
20	Null Object pattern	Jarod42, weston
21	Observer	Arif, user2321864, uzilan
22	Open Close Principle	Mher Didaryan
23	Prototype Pattern	Arif, Jarod42, user2321864
24	Publish-Subscribe	Arif, Jeeter, Stephen C
25	Repository Pattern	bolt19, Leifb
26	Singleton	Bongo, didiz, DimaSan, Draken, fgb, Gul Md Ershad, hellyale, jefry jacky, Loki Astari, Marek Skiba, Mateen Ulhaq, matiaslauriti , Max, Panther, Prateek, RamenChef, Ravindra babu, S.L. Barth , Stephen C, Tazbir Bhuiyan, Tejus Prasad, Vasiliy Vlasov, volvis
27	SOLID	Bongo
28	Static factory method	abbath, Bongo
29	Strategy Pattern	Aseem Bansal, dimitrisli, fabian, M.S. Dousti, Marek Skiba, matiaslauriti, Ravindra babu, Shog9, SjB, Stephen C, still_learning, uzilan
30	Template Method	meJustAndrew, Ravindra babu
31	Visitor Pattern	Daniel Käfer, Jarod42, Loki Astari, Ravindra babu, Stephen Leppik, Vasiliy Vlasov