Ej 1

1. Se puede usar un builder ya que al tener el objeto sándwich con muchas propiedades/configuraciones complejas, se pueden dar errores al momento de declararlo.
2. Public interface ISandwichBuilder {

Public void reset();

Public void setBread();

Public void setCheese();

Public void setMeat();

Public void setVegetables();

Public void setCondiments;

}

Public class SandwichBuilder: ISandwichBuilder{

Private Sandwich sandwich;

Public SandwichBuilder(){

This.reset()

}

Public void reset(){

This.sandwich=new Sandwich()

}

Public Sandwich getResult(){

Sandwich s = new Sandwich()

S=this.sandiwch

This.reset()

Return s;

}

Public void setBread(String bread){

this.sandwich.bread = bread;

}

…

}

Public class program{

Static void main(String[] args){

SandwichBuilder builder = new SandwichBuilder()

Builder.setBread(“White”)

…

Sandwich s = builder.getResult()

Console.WriteLine(s)

}

}

Ej 2

1. Se puede usar el patrón prototype ya que se copian los datos de un objeto varias veces.

Ej 3

1. Se puede usar el patrón Factory ya que dependiendo del servicio, existe una lógica distinta al momento de enviar el mensaje.
2. Public interface IMessage{

Void sendMessage(String msg);

}

Public class Service{

Public IMessage createMessage();

}

Public class EmailService: Service {

Public IMessage createMessage(){

Return new EmailMessage();

}

} // repetir para el resto de tipos de mensajes

Public class EmailMessage: IMessage {

Public void sendMessage(String msg){

Console.writeline(“Email: “ + msg)

}

} // repetir para el resto de mensajes

Public class messagingApp{

Public void sendMessage(String servicetype, message){

Service service;

If(servicetype==”SMS”){

service=new SMSService()

}else if(servicetype==”Email”){

Service= new EmailService()

}else if(servicetype==”Facebook”){

Service=new FacebookService();

}

IMessage msg=Service.createMessage();

Msg.sendMessage(message)

}

}

Ej 4

1. Se puede usar el patrón prototype ya que se utilizan copias del mismo objeto.
2. Public Interface Prototype{

Public Prototype clone();

}

Public class Book: Prototype {

…

Public Prototype clone{

Return new Book(this);

}

}

Public class program{

Public static void main(string[] args){

Book originalBook = new Book(“HP”,”JK”);

Book additionalCopy = originalBook.clone()

…

}

}

Ej 5

1. Al usar un único constructor en el cual se le deben pasar muchos parámetros, se puede usar el patrón Builder.
2. Public interface Builder{

Public void reset();

Public void setFlight(); // repetir para todos los parámetros

Public TravelPlan getResult();

}

Public class TravelPlanBuilder: Builder{

Private TravelPlan travelPlan;

Public TravelPlanBuilder(){

This.reset()

}

Public void reset(){

This.travelPlan=new TravelPlan()

}

Public void setFlight(String flight){

This.travelPlan.flight=flight;

} // repetir para todos

Public TravelPlan getResult(){

TravelPlan tp = new TravelPlan;

Tp = this.travelPlan

This.reset()

Return tp;

}

}

Ej 6

1. Como se debe acceder a un mismo recurso desde varias partes de la aplicación, se puede usar el patrón singleton para que siempre se utilice la misma instancia de la API.
2. Public class Singleton{

Private static API instance;

Private Singleton(){

// codigo para conectarse a la api

}

Public static getInstance(){

If(instance==null){

Instance= new Singleton()

}

Return instance;

}

}