**1ra Iteración**

//Prendas

class Prenda{

*TipoPrenda* tipo

*String* material

*String* colorPrincipal

*String* colorSecundario

    /\*

        Aquí va el constructor con los campos obligatorios para que no

        se generen prendas sin rellenar estos

    \*/

}

//Tipo de prendas

class TipoPrenda{

*Categoria* categoria

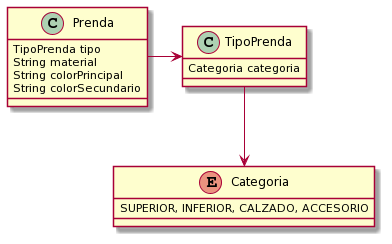
}

//Categorias de prendas

enum Categoria{

    SUPERIOR, INFERIOR, CALZADO, ACCESORIO

}



**2da Iteración**

//Prendas

class BorradorPrenda{

*TipoPrenda* tipo

*Material* material

*Trama* trama = Trama.LISA

*String* colorPrincipal

*String* colorSecundario

    /\*

        Tomo la idea de builder debido a que debemos crear una prenda por

        partes. Para que no se genere una prenda inconsistente entre medio

        es mejor utilizar este borrador. El constructor del builder tendrá

        unicamente al tipo de prenda (que es la primera parte que se

        creará de la misma).

    \*/

*void* setearColorPrincipal(*String* color){

        colorPrincipal = color

    }

*void* setearColorSecundario(*String* color){

        colorSecundario = color

    }

*void* setearMaterial(*String* material){

        //Realizo la validación con el tipo de prenda para ver si es consistente

        this.material = material

    }

*void* setearTrama(*Trama* trama){

        this.trama = trama

    }

*boolean* esUnaPrendaValida(){

        //Valido que los campos obligatorios sean distintos de NULL

    }

*void* generarPrenda(){

        //Verifico si es válida y genero la prenda

        new Prenda(...)

        //Debería almacenarla en algún lugar que sea de utilidad

    }

}

//Tipo de prendas

class TipoPrenda{

*boolean* soportaMaterial(*Material* *material*){

        return materialesConsistentes.contain(material)

    }

}

//Material de una prenda

enum Material{

}

//Trama de una prenda

enum Trama{

    LISA, RAYADA, LUNARES, CUADROS, ESTAMPADO

}

//Uniformes

class Uniforme{

*Prenda* prendaSuperior

*Prenda* prendaInferior

*Prenda* calzado

}

//Instituciones

interface Institucion{

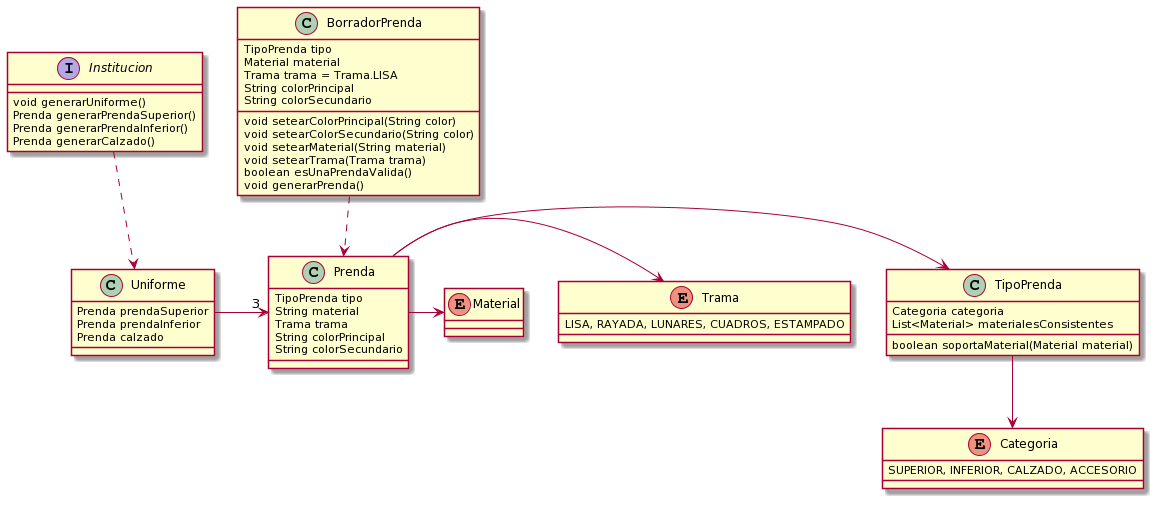
*void* generarUniforme()

*Prenda* generarPrendaSuperior()

*Prenda* generarPrendaInferior()

*Prenda* generarCalzado()

}



**4ta Iteración**

//Prendas

class BorradorPrenda{

*void* setearTemperaturaMaxima(*double* temperatura){

        temperaturaMaxima = temperatura

    }

}

class Prenda{

*double* temperaturaMaxima

}

//Asesor de Imagen

class AsesorDeImagen{

*List*<*Prenda*> prendas

*int* maximaCantidadPrendasPorCategoria

    /\*

        Asumimos que las sugerencias pueden modificarse, una alternativa a esto

        es directamente validar en el constructor antes de generar la

        sugerencia

    \*/

*void* agregarPrenda(*Prenda* *prenda*){

        /\*

            Valido que no haya una cantidad de prendas de esa categoría

            superior al maximo

            Valido tambien que la temperatura actual (obtenida de la clase

            Clima sea menor a la temperatura máxima de la prenda a añadir)

        \*/

        prendas.add(prenda)

    }

*void* quitarPrenda(*Prenda* *prenda*){

        prendas.remove(prenda)

    }

*int* prendasCategoria(*Categoria* *categoria*){

        return prendas.filter(prenda *->* prenda.categoria() == categoria).size()

    }

}

//Servicio Meteorológico

class Clima{

    /\*

        Podemos modelar al clima como singleton y pasar una ciudad a sus

        métodos cada vez que queremos consultar

    \*/

*double* getTemperatura(*String* *ciudad*, *ServicioMeteorologico* *servicio*){

        return servicio.temperatura(ciudad)

    }

*double* getProbabilidadDePrecipitacion(*String* *ciudad*, *ServicioMeteorologico* *servicio*){

        return servicio.probabilidadPrecipitaciones(ciudad)

    }

}

interface ServicioMeteorologico{

*List*<*Map*<*String*,*Object*>> getWeather(*String* *ciudad*)

*double* temperatura()

*double* probabilidadPrecipitaciones()

}

class AccuWeatherAPI implements *ServicioMeteorologico*{

*int* usosActuales

*int* usosMaximos

    /\*

        Generamos estas variables para validar que no se pase de los usos máximos

        que permite la API antes de comenzar a generar gastos.

        Actuaría como una clase static (al igual que el clima).

    \*/

*List*<*Map*<*String*,*Object*>> getWeather(*String* *ciudad*){

        //Verifico que los usos actuales sean menor o igual a los usos máximos

    }

*double* temperatura(*String* *ciudad*){

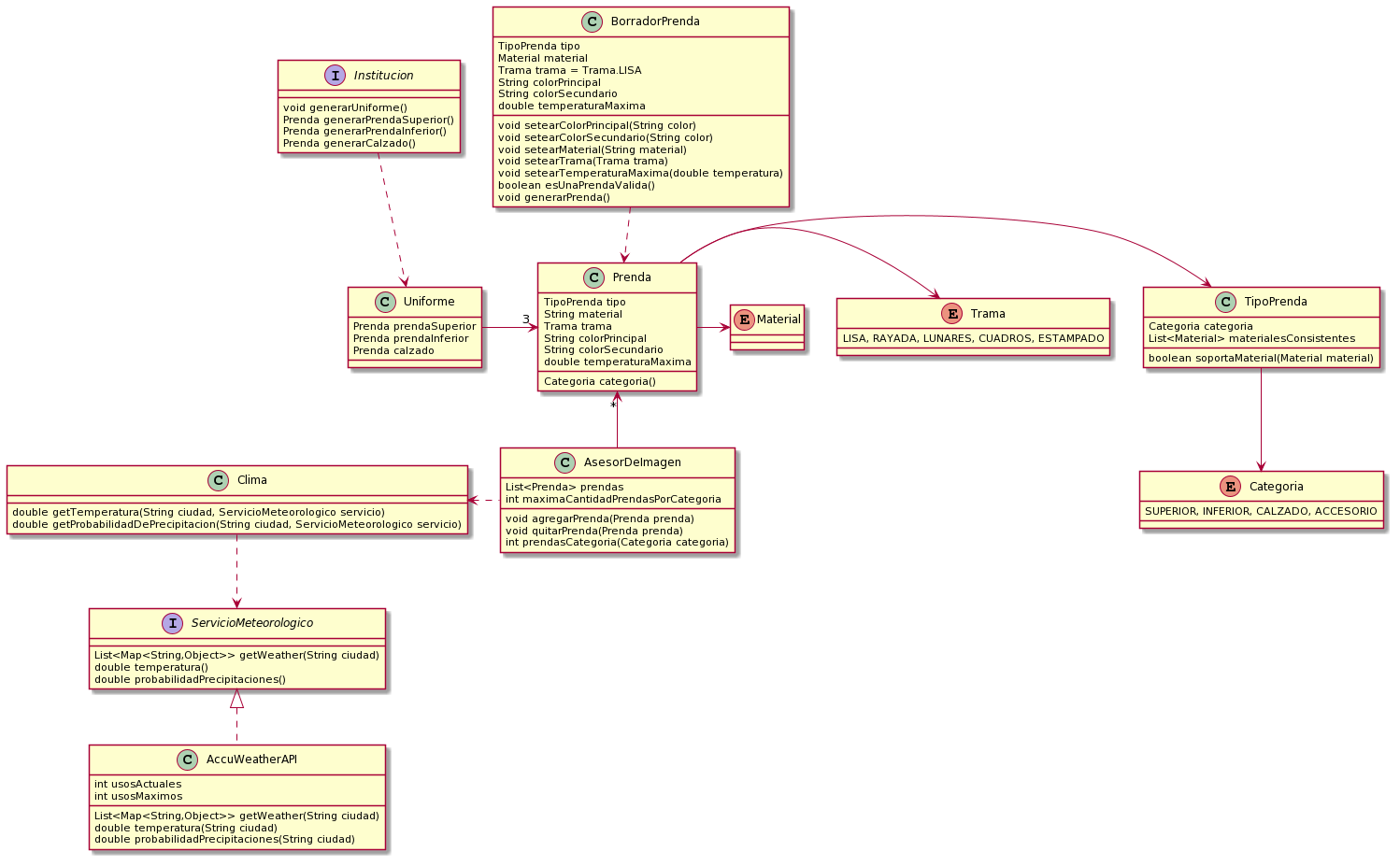
        return this.getWeather().get( 0 ).get( "Temperatura" );

    }

*double* probabilidadPrecipitaciones(*String* *ciudad*){

        return this.getWeather().get( 0 ).get( "PrecipitationProbability" );

    }

}

**5ta Iteración**

//Guardarropas

class Guardarropa{

*List*<*Prenda*> prendas

*Usuario* dueño

*List*<*Usuario*> usuariosConAcceso

*List*<*Propuesta*> propuestas

*void* agregarPrenda(*Prenda* *prenda*){

        prendas.add(prenda)

    }

*void* removerPrenda(*Prenda* *prenda*){

        prendas.remove(prenda)

    }

*void* nuevaPropuesta(*Propuesta* *propuesta*){

        propuestas.add(propuesta)

    }

*void* aceptarPropuesta(*Propuesta* *propuesta*){

        //Valido que la propuesta se encuentre en la lista

        propuesta.ejecutar(this)

        //No la elimino de la lista, para poder deshacerla en algún momento

    }

*void* deshacerPropuesta(*Propuesta* *propuesta*){

        //Valido que la propuesta se encuentre en la lista

        propuesta.deshacer(this)

        /\*

            No la elimino porque asumo que luego puede volver a gustarme,

            y si considero que no lo hará la rechazo

        \*/

    }

*void* rechazarPropuesta(*Propuesta* *propuesta*){

        //Valido que la propuesta se encuentre en la lista

        propuestas.remove(propuesta)

    }

}

//Usuarios

class Usuario{

*void* propuestaParaAgregarPrenda(*Prenda* *prenda*, *Guardarropa* *guardarropa*){

        //Valido que el usuario sea dueño o tenga acceso al guardarropas

        guardarropa.nuevaPropuesta(new AgregarPrenda(prenda))

    }

*void* propuestaParaQuitarPrenda(*Prenda* *prenda*, *Guardarropa* *guardarropa*){

        //Valido que el usuario sea dueño o tenga acceso al guardarropas

        guardarropa.nuevaPropuesta(new RemoverPrenda(prenda))

    }

}

//Propuestas

abstract class Propuesta{

*Prenda* prenda

    abstract *void* ejecutar(*Guardarropa* *guardarropa*)

    abstract *void* deshacer(*Guardarropa* *guardarropa*)

}

class AgregarPrenda extends *Propuesta*{

*void* ejecutar(*Guardarropa* *guardarropa*){

        guardarropa.agregarPrenda(prenda)

    }

*void* deshacer(*Guardarropa* *guardarropa*){

        guardarropa.removerPrenda(prenda)

    }

}

class RemoverPrenda extends *Propuesta*{

*void* ejecutar(*Guardarropa* *guardarropa*){

        guardarropa.removerPrenda(prenda)

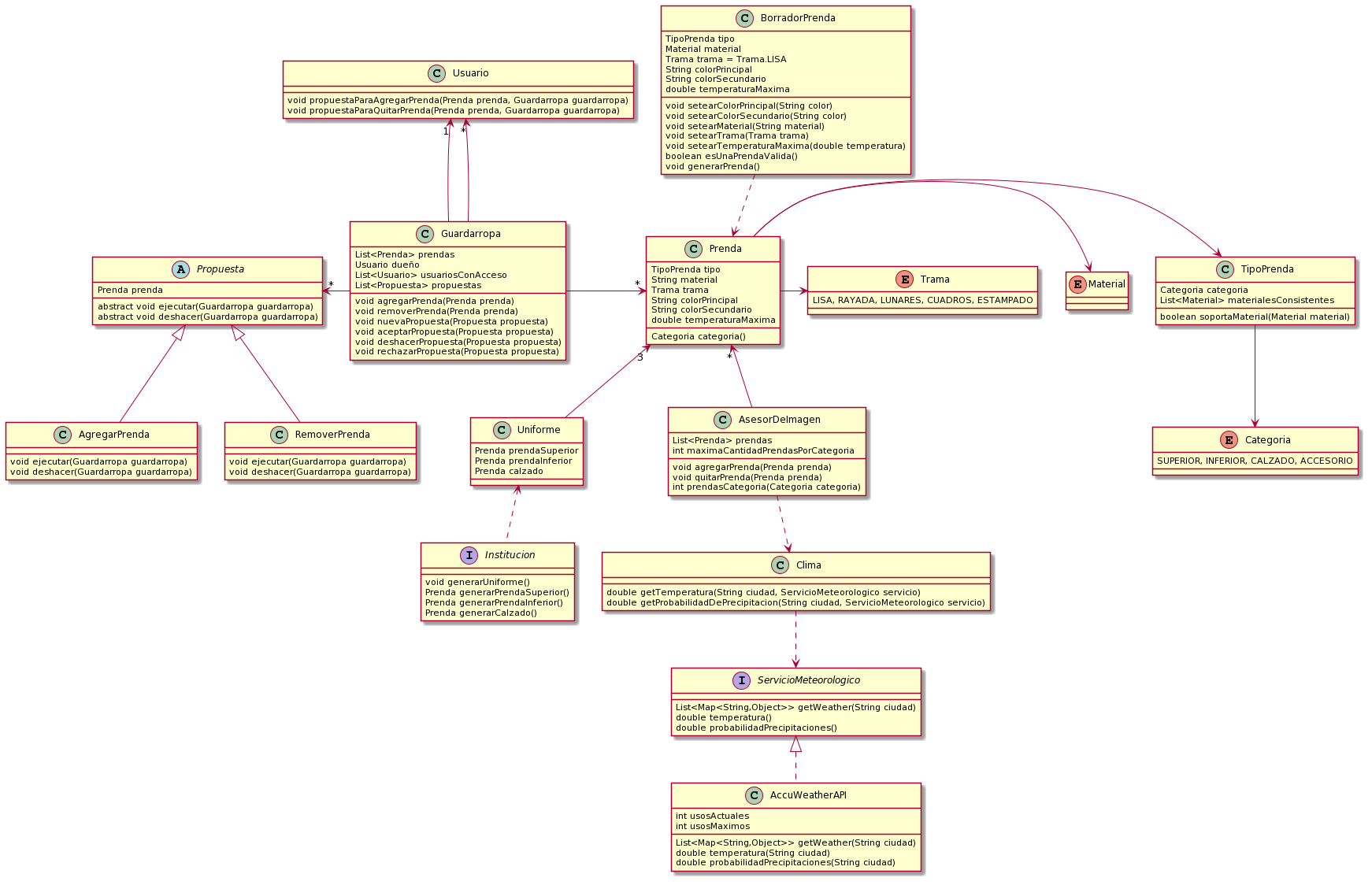
    }

*void* deshacer(*Guardarropa* *guardarropa*){

        guardarropa.agregarPrenda(prenda)

    }

}



**6ta Iteración**

//Asesor de Imagen

class RepositorioSugerencias{

    /\*

        Puedo modelar a la clase como estática para que directamente se llame

        el repo de sugerencias cuando sea necesario.. sin instanciaciones

    \*/

*List*<*Sugerencia*> sugerencias

*Sugerencia* getSugerenciaRandom(){

        //Algoritmo para retornar un elemento random de la sugerencia

    }

}

//Servicio Meteorológico

class Clima{

    /\*

        Modelaremos al clima como clase estática para llamarlo directamente.

        Esto funcionará por el momento, ya que únicamente nos interesa el

        clima de la ciudad de Buenos Aires.

    \*/

*List*<*AlertaMeteorologica*> alertasMeteorologicas

*void* actualizarAlertasMeteorologicas(){

        alertasMeteorologicas = new AccuWeatherAPI().alertasMeteorologicas("Buenos Aires")

    }

*List*<*AlertaMeteorologica*> getAlertasMeteorologicas(){

        return alertasMeteorologicas

    }

*double* getTemperatura(){

        return new AccuWeatherAPI().temperatura("Buenos Aires")

    }

*double* getProbabilidadDePrecipitacion(){

        return new AccuWeatherAPI().probabilidadPrecipitaciones("Buenos Aires")

    }

}

interface ServicioMeteorologico{

*List*<*AlertaMeteorologica*> alertasMeteorologicas()

}

class AccuWeatherAPI implements *ServicioMeteorologico*{

*List*<*AlertaMeteorologica*> alertasMeteorologicas(*String* *ciudad*){

*AccuWeatherAPI* apiClima = new AccuWeatherAPI();

*Map*<*String*, *Object*> alertas = apiClima.getAlertas(ciudad);

      return adaptarAlertasMeteorologicas(alertas.get("CurrentAlerts"));

    }

*List*<*AlertaMeteorologica*> adaptarAlertasMeteorologicas(*List*<*String*> *alertas*){

        //Retorno la conversión de strings al enum correspondiente

    }

}

//Alertas meteorológicas

enum AlertaMeteorologica{

    GRANIZO, TORMENTA

}

//Usuarios

class ListaUsuarios{

    /\*

        En este caso podríamos modelarlo como un singleton, ya que en este

        sistema no habrá distintos grupos de usuarios que pertenezcan a

        diferentes categorizaciones

    \*/

*List*<*Usuario*> usuarios

*void* actualizarSugerenciaDiaria(){

        usuarios.foreach(usuario *->* usuario.pedirSugerencia())

    }

}

class Usuario{

*Sugerencia* sugerenciaDiaria

*List*<*Acciones*> accionesConfigurables

*void* pedirSugerencia(){

        //Obtención de la sugerencia

        sugerenciaDiaria = RepositorioSugerencias.getSugerenciaRandom()

        accionSobreAlertas(Clima.getAlertasMeteorologicas())

    }

*void* accionSobreAlertas(*List*<*AlertaMeteorologica*> *alertas*){

        alertas.foreach(

            alerta *->* accionesConfigurables.foreach(

                        accion.nuevaAlerta(alerta, this)

                      )

        )

    }

}

//Acciones del usuario

interface AccionUsuario{

*void* nuevaAlerta(*AlertaMeteorologica* *alerta*, *Usuario* *usuario*)

}

class EnviarNotificacion implements *AccionUsuario*{

*void* nuevaAlerta(*AlertaMeteorologica* *alerta*, *Usuario* *usuario*){

        if(alerta == AlertaMeteorologica.TORMENTA){

            notificar("Lleva paraguas")

        }

        else if(alerta == AlertaMeteorologica.GRANIZO){

            notificar("Evita salir en auto")

        }

    }

*void* notificar(*String* *texto*){

        new NotificationService().notify(texto)

    }

}

class EnviarMail implements *AccionUsuario*{

*void* nuevaAlerta(*AlertaMeteorologica* *alerta*, *Usuario* *usuario*){

        new MailSender().send(usuario.mail(), "Se ha obtenido la alerta: " + alerta)

    }

}

//Tareas programadas

class PlanificadorAlertasMeteorologicas{

*void* actualizarAlertasMeteorologicas(){

        Clima.actualizarAlertasMeteorologicas()

    }

}

class PlanificadorSugerencias{

*ListaUsuarios* listaDeUsuarios

*void* realizarSugerenciasDiarias(){

        listaDeUsuarios.realizarSugerenciaDiaria()

    }

}

/\*

    Ambos planificadores deben ser objetos globales y se les deberá configurar

    el crontab adecuado para ejecutarse. En este caso, debería ejecturase

    siempre primero el planificador de alertas meteorológicas ya que

    una vez se actualiza la sugerencia diaria del usuario tambien se evalúan

    las acciones a realizar sobre las nuevas alertas.

\*/

