Documento de Diseño de Software (SDD)

para

Simulador de expansión de virus

Versión 1.0

Sergio Flor

Introducción a la Ingeniería de Software.
Prácticas curso 2019/20

16 de abril de 2020

Tabla de Contenido

Historial de Revisiones

Nombre	Fecha	Razón de los cambios	Versión

1. Introducción

1.1 Objetivo

El objetivo de esta práctica es la consolidación de los conocimientos adquiridos con el estudio de la asignatura mediante un ejercicio que abarca los contenidos del curso, en este caso la realización de un programa orientado a objetos en Java que simule la expansión de un virus en una población siguiendo un modelo exponencial. En esta ocasión, se plasma en este documento el proceso de diseño del programa referido.

Un objetivo secundario es comparar como se aborda la realización de un proyecto software desde la óptica de la ingeniería frente al ya conocido y realizado desde la óptica del desarrollador o el programador.

1.2 Ámbito

Además de lo referido en el documento SRD, se puede añadir que desde el punto de vista del diseño, el programa realiza la simulación mediante tres procesos básicos:

- Recoger los parámetros introducidos por el usuario para crear las estructuras de datos y configurar la simulación en base a éstos.
- Realizar los cálculos necesarios para obtener el resultado de la simulación.
- Mostrar los resultados por pantalla para permitir el análisis de los mismos.

1.3 Definiciones, siglas y abreviaturas

Las ya descritas en el Documento de especificación de requisitos(SRD).

1.4 Referencias

Enunciado de las prácticas curso 2019-2020, asignatura de Introducción a la Ingeniería de Software.

Gómez Palomo, S., & Moraleda Gil, E. (2020). *Aproximación a la Ingeniería de Software* (segunda ed.). Madrid: Ramon Areces.

1.5 Panorámica del documento

El resto del documento recoge una visión general de los requisitos funcionales, la descomposición del sistema aplicando la metodología de diseño utilizada y la descripción de estos componentes, para terminar con la comprobación de la coherencia del diseño con la matriz de requisitos/componentes.

2. Panorámica del sistema

2.1 Requisitos funcionales

2.1.1 El programa debe realizar la simulación sobre unas comunidades introducidas por usuario, para lo que contemplará la función:

2.1.1.1 Crear comunidad: Crea y graba las comunidades y sus poblaciones, introducidas por el usuario.

2.1.2 El enunciado menciona la especialización de comunidades en países, provincias o pueblos (como inciso aclaratorio entre paréntesis) aunque describe todas las operaciones de la simulación utilizando comunidades "genéricas". Por tanto, parece recomendable dotar al programa de una estructura básica y funciones que atiendan esta especialización para facilitar el desarrollo de este requerimiento cuando sea necesario. ***(recomendable)***

2.1.3 El programa debe realizar las siguientes funciones para los cálculos de la simulación:

2.1.3.1 Calcular contagio interno: Aplica la fórmula de cálculo de contagio interno en una comunidad para un día.

2.1.3.2 Calcular contagio viajeros: Aplica la fórmula de contagio por viajeros de otras comunidades para un día.

2.1.3.3 Calcular total: Suma los resultados diarios provenientes de las funciones parciales anteriores para todos los días de la simulación, establecidos por la entrada, y controla el rebose de infectados respecto a la población.

2.1.4 El programa debe devolver una tabla con los resultados de infectados y los porcentajes respecto a la población de cada comunidad y del total de todas, día a día. Para lo que realizará las funciones:

2.1.4.1 Crear tabla: Configura la tabla para la representación.

2.1.4.2 Mostrar simulación: Configura la muestra por pantalla de la representación de las simulaciones.

2.1.5 El programa debe permitir al usuario introducir los datos requeridos y realizar modificaciones sobre los parámetros, por lo que debe ofrecerse una interfaz adecuada para poder realizar estas operaciones facilitando la modificación de un o unos parámetros y la conservación de los valores previos en otros.

2.1.6 La asignación del porcentaje de habitantes de cada comunidad que viaja a otras comunidades se introduce igual para simplificar, lo que implica que sería deseable en una versión más compleja poder trabajar con distintos porcentajes para cada comunidad, y por tanto permitir su introducción.***(opcional)***

2.1.7 El programa debe presentar los datos de forma que se puedan analizar de forma pausada, para ello puede ser deseable permitir recuperar los resultados de las distintas simulaciones realizadas. ***(opcional)***

2.1.8 Se aprecia también presentar el resultado de forma gráfica para lo que el programa debería de realizar gráficos que representan la evolución diaria de los distintos datos que se piden, pudiéndose mostrar por ejemplo, en dos gráficas que agrupen las comunidades y el total de la evolución de infectados por un lado y de la evolución de porcentajes por otro. ***(recomendable)***

3. Contexto del sistema

No existe conexión con otros sistemas

4. Diseño del sistema

4.1 Metodología de diseño de alto nivel

Se utilizará una metodología de diseño orientada a objetos. En todo momento se han tenido en cuenta los conceptos fundamentales de esta metodología y se ha perseguido el encapsulamiento y ocultación de los datos, utilizar la herencia donde ha sido posible al igual que el polimorfismo tanto en el paso de parámetros como en la sobrecarga de métodos y, por supuesto, la cohesión en el diseño de las clases para conseguir el menor acoplamiento posible.

4.2 Descomposición el sistema

De los procesos, almacenes, funciones y resto de elementos descritos en las especificaciones del sistema, se deriva una jerarquía de clases que deberá darles correspondencia. De hecho, podemos ver como la relación entre estas clases corresponde con los 3 procesos del DFD 0, pudiendo agruparlas según se relacionen con las comunidades, con el cálculo de la simulación o con la muestra de los resultados.

Así pues, tenemos:

4.2.1 Una clase principal **Main** que representa el proceso principal y que inicia y desarrolla ordenadamente el resto de procesos.

4.2.2 Una clase **Almacen_Comunidades** correspondiente a dicho almacén, que será único, y que por tanto encaja con la implementación de un patrón Singleton. Además, se requiere:

- La creación de la colección de comunidades con los datos introducidos por el usuario.
- Ofrecer un método para recuperar la colección guardada.

4.2.3 Una clase **Comunidad** que modele la estructura de los datos requeridos: población, porcentaje de viajeros y un nombre identificativo; y de la que se requieren además de su creación los siguientes métodos:

- Modificar los datos.
- Recuperar los datos.
- Calcular y devolver el número de viajeros según el porcentaje asignado.

4.2.4 Atendiendo al requisito que preveía la especialización de la comunidad en países, provincias y pueblos, tres subclases con dichos nombres que extiendan a la clase Comunidad, que queda como modelo genérico de comunidad. Estas tres clases deben poder modelar la relación obvia existente entre ellas: un **País** está formado por **Provincias** y una provincia por **Pueblos**. Para gestionar esta relación las clases deben ofrecer métodos para:

- Asociar pueblos a provincias y provincias a países.
- Recuperar los pueblos de una provincia y las provincias de un país.

- Sobrescribir la recuperación de la población de forma acorde a la lógica de esta relación.

4.2.5 Dada esta especialización, es apropiado añadir un tipo **enumerado** que represente los tipos de comunidades modeladas, y añadir un método a la superclase que devuelva el tipo de comunidad.

4.2.6 Una clase **Simulador**, que represente el proceso principal de configurar y calcular la simulación, y que debe almacenar el valor de los parámetros de la simulación e implementar métodos de cálculo para:

- Calcular el total de infectados para cada comunidad y día de los introducidos como totales, gestionando el proceso y sumando el resultado diario interno y de viajeros, delegado en los otros dos métodos que no requerirán por tanto acceso público.
- El cálculo del resultado interno de un día, a partir del resultado anterior.
- El cálculo del resultado por viajeros de un día, a partir del número de viajeros y los resultados del día anterior del resto de comunidades.

4.2.7 Una clase **Almacen_Simulaciones**, correspondiente a dicho almacén, que al igual que el almacén de comunidades, encaja con un patrón Singleton, y por tanto se usará esta implementación. Se ocupa de guardar y gestionar los resultados de las simulaciones para su consulta. Debe por tanto almacenarlos de forma que se pueda pasar ordenadamente de uno a otro, como por ejemplo una lista con acceso por punto de interés, y ofrecer los métodos para:

- Añadir resultados al almacén.
- Recuperar el resultado actual, el anterior y el siguiente.

4.2.8 Una clase **Resultado_Simulación**, que modela la estructura de datos de los resultados: una serie de comunidades, cada una con los resultados para cada fecha de los días de la simulación. Podría utilizarse un array para esta estructura, pero desde el punto de vista lógico y funcional se ajusta de forma más apropiada una estructura de tipo mapa anidado. Se requieren de los resultados, además, métodos para:

- Añadir un resultado de una comunidad para un día concreto.
- Recuperar los resultados totales para un día de una comunidad o del total.
- Recuperar el porcentaje de infectados para un día de una comunidad o del total.

4.2.9 Una clase que se denominará **Interfaz_Grafica**, encargada principalmente de ofrecer la interfaz al usuario a la hora de introducir o modificar datos de entrada, y de actualizar la representación de los resultados por pantalla, para lo que es conveniente implementar mediante un patrón Observer que se relacione con la tabla y los gráficos que se generen sobre el resultado actual.

4.2.10 Una clase **TablaDatos**, encargada de:

- Crear y configurar la presentación de la tabla de resultados.
- Recuperar la tabla de resultados.

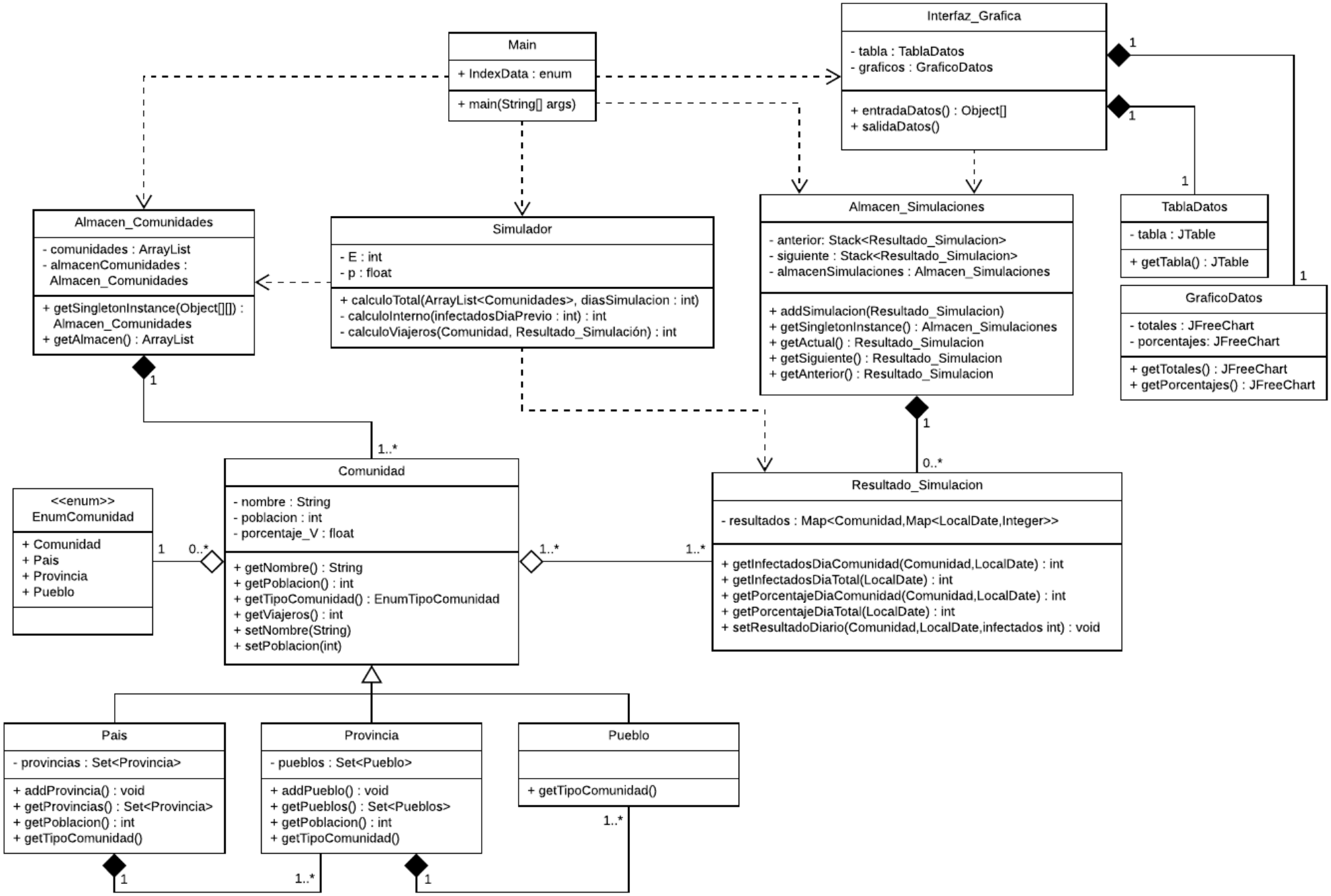
4.2.11 Una clase **GráficaDatos**, de la que se requerirá:

- Crear y configurar las presentaciones de los resultados con gráficos.
- Recuperar los gráficos de resultados.

Una vez descrita la jerarquía de clases, se expone el modelo diseñado mediante el diagrama de clases UML de la siguiente página. Este diagrama parte de la descomposición inicial detallada hasta este punto,

pero se ha ido refinando conforme se han realizado las tareas diseño de las clases descritas en la siguiente sección, por lo que aparecen ya con los atributos y métodos finales del diseño.

Por claridad, se han omitido los nombres de algunos parámetros cuando el nombre de la clase o tipo al que pertenece resulta suficientemente claro para la comprensión del diagrama.


Main
+ IndexData : enum
+ main(String[] args)
Interfaz_Grafica
- tabla : TablaDatos
- graficos : GraficoDatos
+ entradaDatos() : Object[]
+ salidaDatos()
Almacen_Comunidades
- comunidades : ArrayList
- almacenComunidades : Almacen_Comunidades
+ getSingletonInstance(Object[][]) : Almacen_Comunidades
+ getAlmacen() : ArrayList
Simulador
- E : int
- p : float
+ calculoTotal(ArrayList<Comunidades>, diasSimulacion : int)
- calculoInterno(infectadosDiaPrevio : int) : int
- calculoViajeros(Comunidad, Resultado_Simulación) : int
Almacen_Simulaciones
- anterior: Stack<Resultado_Simulacion>
- siguiente : Stack<Resultado_Simulacion>
- almacenSimulaciones : Almacen_Simulaciones
+ addSimulacion(Resultado_Simulacion)
+ getSingletonInstance() : Almacen_Simulaciones
+ getActual() : Resultado_Simulacion
+ getSiguiente() : Resultado_Simulacion
+ getAnterior() : Resultado_Simulacion
TablaDatos
- tabla : JTable
+ getTabla() : JTable
GraficoDatos
- totales : JFreeChart
- porcentajes: JFreeChart
+ getTotales() : JFreeChart
+ getPorcentajes() : JFreeChart
<<enum>>
EnumComunidad
+ Comunidad
+ Pais
+ Provincia
+ Pueblo
Comunidad
- nombre : String
- poblacion : int
- porcentaje_V : float
+ getNombre() : String
+ getPoblacion() : int
+ getTipoComunidad() : EnumTipoComunidad
+ getViajeros() : int
+ setNombre(String)
+ setPoblacion(int)
Resultado_Simulacion
- resultados : Map<Comunidad,Map<LocalDate,Integer>>
+ getInfectadosDiaComunidad(Comunidad,LocalDate) : int
+ getInfectadosDiaTotal(LocalDate) : int
+ getPorcentajeDiaComunidad(Comunidad,LocalDate) : int
+ getPorcentajeDiaTotal(LocalDate) : int
+ setResultadoDiario(Comunidad,LocalDate,infectados int) : void
Pais
- provincias : Set<Provincia>
+ addProvincia() : void
+ getProvincias() : Set<Provincia>
+ getPoblacion() : int
+ getTipoComunidad()
Provincia
- pueblos : Set<Pueblo>
+ addPueblo() : void
+ getPueblos() : Set<Pueblos>
+ getPoblacion() : int
+ getTipoComunidad()
Pueblo
+ getTipoComunidad()
1
0..*
1..*

5. Descripción de componentes

5.1 Clase Main

5.1.1 Tipo: Clase principal.

5.1.2 Objetivo: Gestionar el funcionamiento y coordinar los componentes de la aplicación.

5.1.3 Función: Iniciar el programa, realizar el diálogo con el usuario y desarrollar la secuencia lógica del programa.

5.1.4 Subordinados: Interfaz_Grafica, Almacen_Comunidades, Simulador, Almacen_Simulaciones.

5.1.5 Dependencias: Ninguna.

5.1.6 Interfases:

- Los datos recuperados desde Interfaz_Grafica, corresponden a los DATOS_COMUNIDAD y DATOS_SIMULACIÓN (ver D.D. SRD) se empaquetarán en una matriz bidimensional polimórfica de la clase Object, Object[ ][ ]. Se almacenará un tipo enumerado en la clase Main que indexe la posición de cada tipo de dato

Posición	Tipo	Dato contenido
[0][i]	String	Nombre de una comunidad
[1][i]	Integer	Población de una comunidad
[2][0]	Integer	Porcentaje de viajeros
[3][0]	Integer	Parámetro E
[4][0]	Integer	Parámetro p
[5][0]	Integer	Número de días a simular
[6][0]	LocalDate	Fecha inicial

- A las clases Almacen_Comunidades y Simulador se les proveerá de los datos que requieran para construir sus instancias con esta matriz pasada por parámetro.
- La creación de un Almacen_Simulaciones no requiere más que la llamada la instancia.
- La recepción desde la clase Simulador y la adición al almacén de los resultados, se realiza con el paso por parámetro del objeto Resultado_Simulación correspondiente.

5.1.7 Recursos:

- Paquete java.time para el manejo de fechas.

5.1.8 Referencias: Ninguna.

5.1.9 Procesos:

Crear *almacenSimulaciones*

Recibir los datos de entrada

REPETIR

Crear *almacenComunidades*

Crear un Simulador con los parámetros introducidos

Obtener el resultado llamando al cálculo de la simulación

Añadir el resultado a *almacenSimulaciones* como resultado actual

Marcar en la interfaz la existencia de un resultado para mostrar por pantalla

SI se modifican los parámetros de entrada

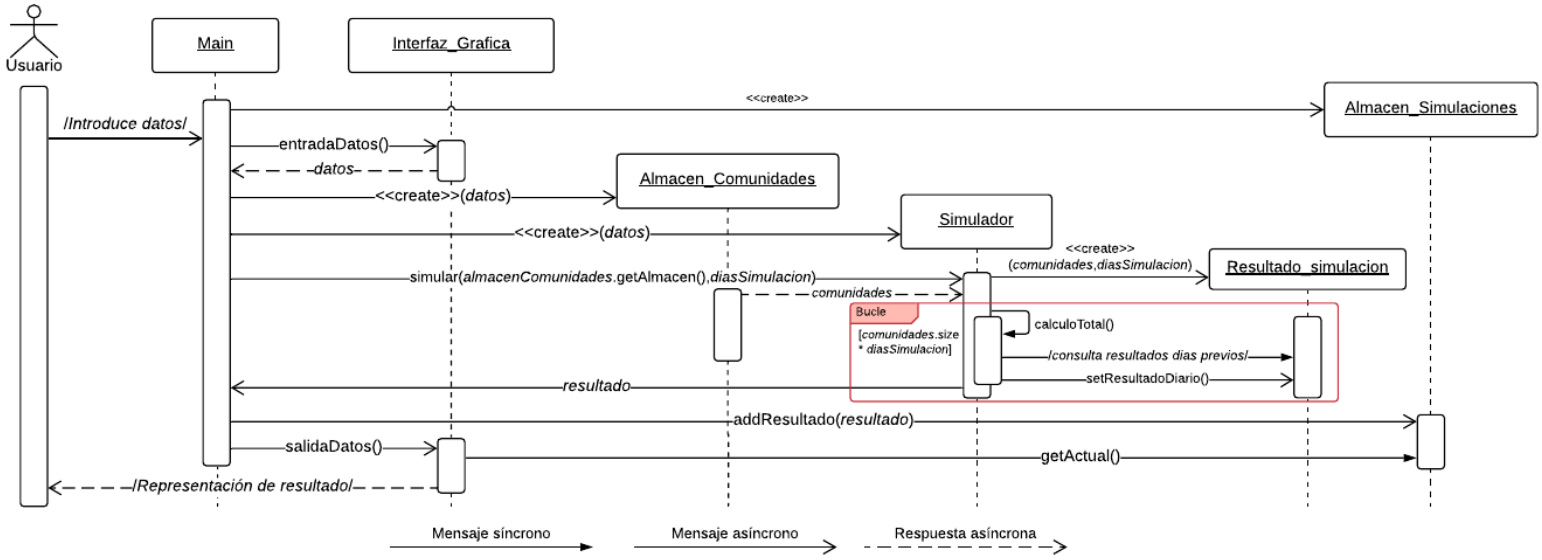
CONTINUAR con la siguiente iteración

FIN-SI

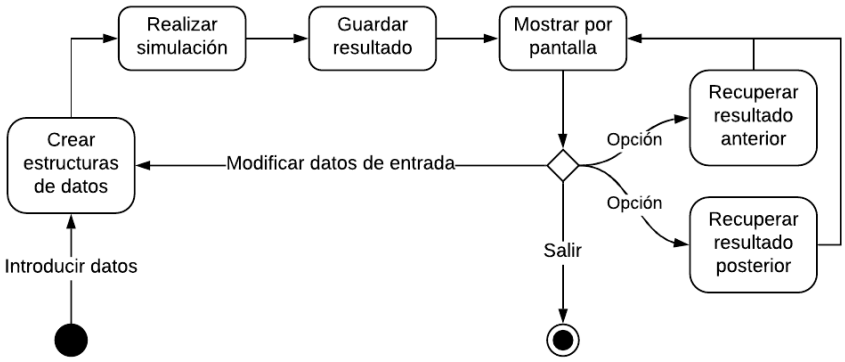
HASTA fin de sesión.

Salir

- En el siguiente diagrama de secuencia UML se representa el proceso realizado al iniciar el programa y realizar la primera simulación:



- El siguiente diagrama de estados UML, resume en el funcionamiento del programa desde el punto de vista del Main y de la gestión de los procesos y estados que se le encomiendan:



5.1.10 Datos públicos:

- IndexData : enum

  { NOMBRE, POBLACION, PORCENTAJE, E, P, DIAS, FECHA }

5.2 Clase Almacen_Comunidades

5.2.1 Tipo: Colección de datos.

5.2.2 Objetivo: Proveer a los métodos de cálculo de la información de las comunidades.

5.2.3 Función: Crea y gestiona una colección de instancias de la clase Comunidad, a partir de los datos introducidos por el usuario.

5.2.4 Subordinados: Main.IndexData, Comunidades

5.2.5 Dependencias: Simulador.

5.2.6 Interfases:

- Recibe de Main la matriz de datos de entrada, de los que extrae los referentes a las comunidades mediante el índice del enumerado Main.IndexData.
- A la llamada del método correspondiente, devuelve la colección de comunidades.

5.2.7 Recursos:

- Paquete java.util para el manejo de colecciones.

5.2.8 Referencias: Ninguna.

5.2.9 Procesos:

- Constructor Almacen_Comunidades(Object[ ][ ] *datos*)

  El constructor recibe por parámetro un objeto *datos*

  PARA i DESDE 0 HASTA *datos*[Main.IndexData.NOMBRE].length PASO +1 HACER

  Añadir a *almacenComunidades* nueva Comunidad(

  datos[Main.IndexData.NOMBRE][i],

  datos[Main.IndexData.POBLACION][i],

  datos[Main.IndexData.PORCENTAJE][i]

)

  FIN-PARA

- Selector del patrón getSingletonInstance(Object[ ][ ])

  SI la instancia única == null ENTONCES

  Llamada a constructor

  FIN-SI

  DEVOLVER instancia única

- Selector getAlmacen()

DEVOLVER colección de comunidades

5.2.10 Datos públicos: Ninguno.

5.2.11 Datos privados:

- comunidades : ArrayList<Comunidad>
- (estático – instancia única) almacenComunidades : Almacen_Comunidades

5.3 Clase Comunidad

5.3.1 Tipo: Abstracción de datos.

5.3.2 Objetivo: Proveer el tipo abstracto de datos necesarios para las comunidades.

5.3.3 Función: Crear objetos y proporcionar acceso y modificación a sus atributos.

5.3.4 Subordinados: Ninguno.

5.3.5 Dependencias: Almacen_Comunidades.

5.3.6 Interfases: La interacción con el almacén se circunscribe a su creación mediante el constructor y el acceso y modificación de sus datos mediante parámetros correspondientes a sus atributos.

5.3.7 Recursos: Ninguno.

5.3.8 Referencias: Ninguna.

5.3.9 Procesos:

- Constructor plano y constructor con atributos:

  Comunidad(String nombre, int población, int porcentaje)
- Selectores y mutadores de los atributos
- Método getTipoComunidad()

  DEVOLVER EnumComunidad.COMUNIDAD
- Método getViajeros()

  DEVOLVER *poblacion* * (*porcentaje_V* /100)

5.3.10 Datos públicos:

- EnumComunidad : enum = {COMUNIDAD, PAIS, PROVINCIA, PUEBLO}

5.3.11 Datos privados:

- nombre: String
- poblacion : int
- porcentaje_V : int

5.4 Clase Pais *extends* Comunidad

5.4.1 Tipo: Abstracción de datos. Subclase de Comunidad

5.4.2 Objetivo: Especificar el tipo abstracto de datos para los países.

5.4.3 Función: Crear objetos y proporcionar acceso y modificación a sus atributos.

5.4.4 Subordinados: Ninguno.

5.4.5 Dependencias: Ninguna.

5.4.6 Interfases: No aplicable.

5.4.7 Recursos:

- Paquete java.util para el manejo de colecciones.

5.4.8 Referencias: Ninguna.

5.4.9 Procesos:

- Método addProvincia(Provincia provincia)

  añadir provincia a conjunto de provincias
- Selector getProvincia()

  DEVOLVER conjunto de provincias
- Selector (sobrescrito) getPoblacion()

  PARA CADA provincia del conjunto HACER

  provincia.getPoblacion()

  FIN-PARA
- Selector (sobrescrito) getTipoComunidad()

  DEVOLVER EnumComunidad.PAIS

5.4.10 Datos públicos: Ninguno.

5.4.11 Datos privados:

- provincias : HashSet<Provincia>
- población (heredado de Comunidad) = 0

5.5 Clase Provincia *extends* Comunidad

5.5.1 Tipo: Abstracción de datos. Subclase de Comunidad

5.5.2 Objetivo: Especificar el tipo abstracto de datos para las provincias.

5.5.3 Función: Crear objetos y proporcionar acceso y modificación a sus atributos.

5.5.4 Subordinados: Ninguno.

5.5.5 Dependencias: Ninguna.

5.5.6 Interfases: No aplicable.

5.5.7 Recursos:

- Paquete java.util para el manejo de colecciones.

5.5.8 Referencias: Ninguna.

5.5.9 Procesos:

- Método addPueblo(Pueblo pueblo)

  añadir pueblo a conjunto de pueblos
- Selector getPueblos()

  DEVOLVER conjunto de pueblos
- Selector (sobrescrito) getPoblacion()

  PARA CADA pueblo del conjunto HACER

  pueblo.getPoblacion()

  FIN-PARA
- Selector (sobrescrito) getTipoComunidad()

  DEVOLVER EnumComunidad.PROVINCIA

5.5.10 Datos públicos: Ninguno.

5.5.11 Datos privados:

- pueblos: HashSet< Pueblo >
- población (heredado de Comunidad) = 0

5.6 Clase Pueblo *extends* Comunidad

5.6.1 Tipo: Abstracción de datos. Subclase de Comunidad

5.6.2 Objetivo: Especificar el tipo abstracto de datos para los pueblos.

5.6.3 Función: Crear objetos y proporcionar acceso y modificación a sus atributos.

5.6.4 Subordinados: Ninguno.

5.6.5 Dependencias: Ninguna.

5.6.6 Interfases: No aplicable.

5.6.7 Recursos: Ninguno.

5.6.8 Referencias: Ninguna.

5.6.9 Procesos:

- Selector (sobrescrito) getTipoComunidad()

  DEVOLVER EnumComunidad.PUEBLO

5.6.10 Datos públicos: Ninguno.

5.6.11 Datos privados: Ninguno.

5.7 Clase Almacen_Simulaciones

5.7.1 Tipo: Colección de datos.

5.7.2 Objetivo: Proveer a la interfaz de un sistema de navegación por los resultados de simulaciones.

5.7.3 Función: Crea y gestiona una colección de objetos de Resultado_Simulacion, implementada a modo de lista con acceso por punto de interés, usando como estructura dos pilas, en una de cuyas cimas se encuentra el resultado actual.

5.7.4 Subordinados: Resultado_Simulacion.

5.7.5 Dependencias: Main, Interfaz_Gráfica, TablaDatos, GráficoDatos.

5.7.6 Interfases:

- La interacción del resto de componentes con esta clase, se realiza recibiendo o pasando como parámetro un objeto de Resultado_Simulación.
- Para gestionar la actualización de la vista del resultado que se encuentre en la posición actual, mediante tabla y gráficos, conviene implementar un patrón Observer, por lo que en este caso el modelo deberá implementar la interface Observable del paquete java.util, e implementar los métodos para comunicarse con la vista.

5.7.7 Recursos:

- Paquete java.util para el manejo de colecciones y la realización del interfaz Observable.

5.7.8 Referencias: Ninguna.

5.7.9 Procesos:

- El constructor creará en su llamada una estructura de datos vacía
- Método addResultado(Resultado_simulacion *resultado*)

  SI la pila *siguiente* está vacía ENTONCES

  apila en *siguiente* el *resultado*

  SI-NO

  apila en *anterior* la cima de *siguiente* al desapilarla

  apila en *siguiente* el *resultado*

  FIN-SI
- Selector getActual() : Resultado_simulacion

  SI la pila *siguiente* está vacía ENTONCES

  DEVUELVE null

  FIN-SI

  DEVUELVE la cima de *siguiente* sin desapilar
- Selector getSiguiente() : Resultado_simulacion

  SI el tamaño de la pila *siguiente* <= 1 ENTONCES

  DEVUELVE null

  FIN-SI

  apila en *anterior* la cima de *siguiente* al desapilarla

  DEVUELVE llamada a getActual()

- Selector getAnterior() : Resultado_simulacion

  SI la pila *anterior* está vacía ENTONCES

  DEVUELVE null

  FIN-SI

  apila en *siguiente* la cima de *anterior* al desapilarla

  DEVUELVE llamada a getActual()

5.7.10 Datos públicos: Ninguno.

5.7.11 Datos privados:

- anterior : Stack<Comunidad>
- siguiente : Stack<Comunidad>
- (estático – instancia única) almacenSimulaciones : Almacen_Simulaciones

5.8 Clase Resultado_Simulacion

5.8.1 Tipo: Abstracción de datos.

5.8.2 Objetivo: Proveer una estructura de datos para almacenar los resultados de una simulación.

5.8.3 Función: Crear y gestionar el almacenamiento de los resultados mediante mapas anidados, ofreciendo métodos para consultar los datos que se representarán por pantalla y para añadir los resultados de los cálculos.

5.8.4 Subordinados: Comunidad.

5.8.5 Dependencias: Almacen_Simulaciones.

5.8.6 Interfases:

- Mediante el acceso a la estructura se deben ofrecer los datos necesarios para la representación en pantalla, por lo que interesa controlar los límites. Las clases encargadas de mostrar por pantalla los resultados deben poder acceder con seguridad a las fechas iniciales y finales en las llamadas a los getters.

5.8.7 Recursos:

- Paquete java.util para el manejo de colecciones.
- Paquete java.time para el manejo de fechas.

5.8.8 Referencias: Ninguna.

5.8.9 Procesos:

- Mutador setResultadoDiario(Comunidad comunidad, LocalDate fecha, int infectados)

  recuperar mapa asociado a clave *comunidad* y

  añadir relación (clave *fecha* – valor *infectados*)

- Selector getResultadoDiaComunidad(Comunidad comunidad, LocalDate fecha) : int

  recuperar mapa asociado a clave *comunidad* y

DEVOLVER valor asociado a *fecha*

- Selector getPorcentajeDiaComunidad(Comunidad comunidad, LocalDate fecha) : int

  DEVOLVER (llamada a método getResultadoDiaComunidad(*comunidad,fecha)* * 100)

  / (llamada a método *comunidad*.getPoblacion())

- Selector getResultadoDiaTotal(LocalDate fecha) : int

  recuperar en cadena los mapas de resultados de todas las comunidades y

  recuperar en cadena la correspondencia de los mapas con los valores para *fecha* y

  DEVOLVER la suma de todos los valores

- Selector getMuestraTotal() : int

  recuperar en cadena el conjunto de comunidades clave del mapa de resultados y

  recuperar en cadena la correspondencia de cada comunidad con el valor de su población y

  DEVOLVER la suma de todos los valores

- Selector getResultadoDiaTotal(LocalDate fecha) : int

  DEVOLVER (llamada a método getResultadoDiaTotal(*fecha*) * 100)

  / (llamada a método getMuestraTotal())

5.8.10 Datos públicos: Ninguno.

5.8.11 Datos privados:

- resultados : HashMap< Comunidad , HashMap< LocalDate , Integer >>

5.9 Clase Simulador

5.9.1 Tipo: Abstracción funcional.

5.9.2 Objetivo: Realizar los cálculos necesarios para obtener los resultados de una simulación.

5.9.3 Función: Calcula de acuerdo a los parámetros de simulación introducidos por usuario y los datos de las comunidades, los resultados de la simulación.

5.9.4 Subordinados: Almacen_Comunidad, Resultado_Simulacion.

5.9.5 Dependencias: Main.

5.9.6 Interfases:

- Recibe de Main la matriz de datos de entrada, de los que extrae los referentes a la simulación mediante el índice del enumerado Main.IndexData.
- Recibe de Main la llamada a método para calcular el resultado, recibiendo por parámetro la lista de comunidades, la fecha inicial y el número de días para los que se realiza. Alternativamente se podrá implementar un método que sobrecargue a este y utilice la fecha actual por defecto.
- Interacciona con Resultado_Simulacion a través de sus métodos para la realización de los cálculos, obteniendo los resultados de los días previos, o para la anotación de los resultados de los cálculos en las fechas y comunidades correspondientes.

5.9.7 Recursos:

- Paquete java.util para el manejo de colecciones.
- Paquete java.time para el manejo de fechas.

5.9.8 Referencias: Ninguna.

5.9.9 Procesos:

- Método calculoTotal(ArrayList<Comunidad> comunidades, LocalDate fecha, int diasSimulacion)

  crear objeto Resultado_Simulacion *resultado*

  crear variable int *subtotal*

  PARA CADA comunidad de la lista *comunidades* HACER

  llamada a *resultado*.setResultadoDiario(…) y set a 0 los resultados del primer día

  FIN-PARA

  llamada a *resultado*.serResultadoDiario(…)y set a 1 el resultado de la primera comunidad.

  PARA i DESDE 2 HASTA incluido *diasSimulacion* PASO +1 HACER

  incrementar *fecha* en un día

  PARA-CADA comunidad de la lista *comunidades* HACER

  SI numero de infectados del día previo == población de la comunidad

  llamada a *resultado*.setResultadoDiario(…) y set la población

  CONTINUAR siguiente iteración

  FIN-SI

  subtotal = llamada a calculoInterno(recuperar infectados día previo)

  + llamada a calculoViajeros(…)

  llamada a *resultado*.setResultadoDiario(…) y set

  el valor menor entre(*subtotal,* población de la comunidad)

  FIN-PARA

  FIN-PARA

  DEVOLVER *resultado*

- Método calculoInterno(int infectadosDiaPrevio) : int

  DEVOLVER *infectadosDiaPrevio* * (1 + E + p)

- Método calculoViajeros(Comunidad comunidadPropia, Resultado_Simulación resultado, LocalDate diaPrevio) : int

  recuperar en cadena las comunidades incluidas en los resultados y

  filtrar la comunidad propia para eliminarla de la cadena y

recuperar en cadena la correspondencia de las comunidades con el resultado de formular

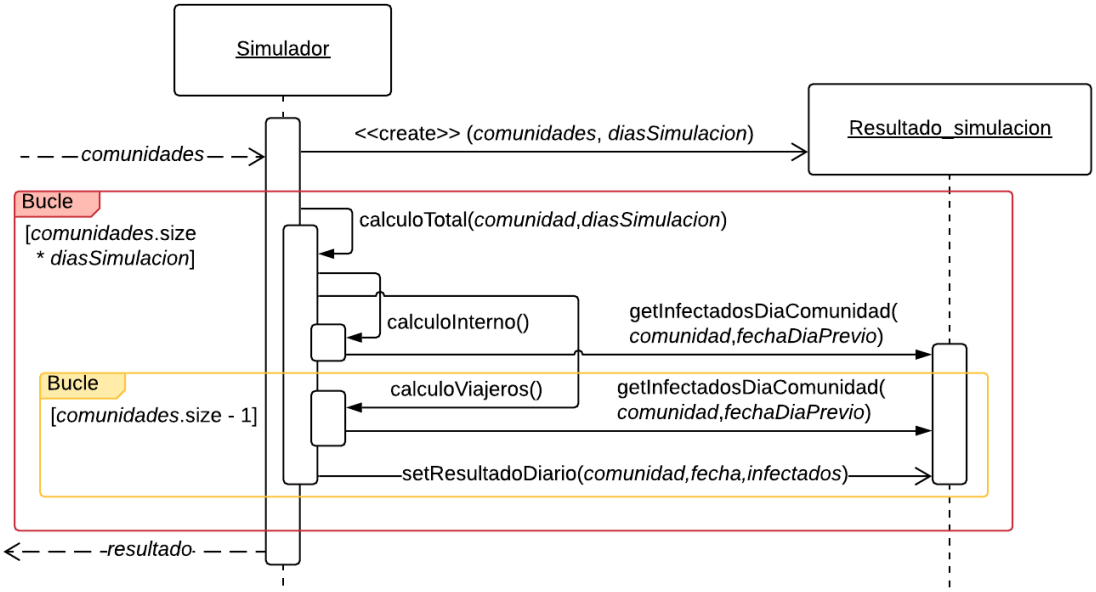
($E * p$ * viajeros de la comunidad

* infectados de la comunidad el día previo

/ población de la comunidad) y

DEVOLVER la suma de los resultados de aplicar la fórmula para cada comunidad

- A continuación se presenta el diagrama de secuencia que incluye todos los procesos del cálculo de la simulación. Se ha omitido por claridad la anotación del resultado del primer día.



5.9.10 Datos públicos: Ninguno.

5.9.11 Datos privados:

- E : int
- p : float /El parámetro se recibe en porcentaje pero se almacena en probabilidad 0-1/

5.10 Clase Interfaz_Grafica

5.10.1 Tipo: Abstracción funcional

5.10.2 Objetivo: Realizar la representación por pantalla de los datos de forma adecuada.

5.10.3 Función: Configurar la interfaz de entrada de datos de usuario y la actualización de tablas y gráficos de acuerdo a la selección de los resultados de simulación por el usuario.

5.10.4 Subordinados: Almacen_Simulaciones, TablaDatos, GraficoDatos.

5.10.5 Dependencias: Main.

5.10.6 Interfases:

- Debe recoger los datos introducidos o modificados por usuario y trasladarlos al Main en la matriz de datos descrita anteriormente.

- Mostrará la tabla de datos y los gráficos del resultado de la simulación situada en la posición "actual" de *almacenSimulaciones* para lo que conviene implementar un patrón Observer, por lo que en este caso la vista, debe implementar la interfaz Observer, y los métodos necesarios para comunicarse con el modelo.

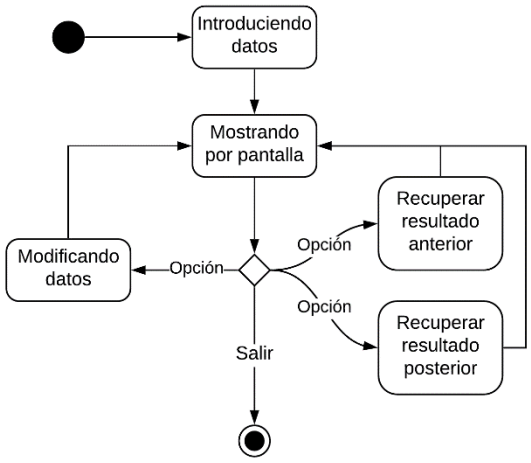
5.10.7 Recursos:

- Paquete java.swing y java.awt para la creación de una interfaz gráfica

5.10.8 Referencias: Ninguna.

5.10.9 Procesos:

- A continuación se presenta un diagrama de estados UML realizado desde el punto de vista de la interfaz, y que muestra las opciones que debe atender y ofrecer la interfaz gráfica:



5.10.10 Datos públicos: Ninguno.

5.10.11 Datos privados:

- tabla : TablaDatos
- gráficos : GraficoDatos

5.11 Clase TablaDatos

5.11.1 Tipo: Abstracción funcional

5.11.2 Objetivo: Obtener la representación de los datos en forma de tabla.

5.11.3 Función: Configurar a partir de los resultados de una simulación, una tabla que muestre los datos requeridos en las especificaciones.

5.11.4 Subordinados: Ninguno.

5.11.5 Dependencias: Interfaz_Grafica.

5.11.6 Interfases:

- Utilizará los selectores de los objetos Resultado_Simulacion para crear la tabla de datos.

5.11.7 Recursos:

- Paquete java.swing y java.awt para la creación de una tabla de la clase JTable.

5.11.8 Referencias: Ninguna.

5.11.9 Procesos:

- Constructor TablaDatos(Resultado_Simulacion simulación)

  crear la matriz de los títulos de columna String[total de comunidades+2] *columnas*

  crear la matriz de datos String[diasSimulacion+2][total de comunidades+2] *datos*

  asignar desde *columnas*[1] a *columnas*[total de comunidades] los nombres de las mismas

  asignar *columnas*[total de comunidades +1] al resultado total

  asignar a las posiciones de *datos*[1][1] a *datos*[1][total de comunidades] las poblaciones

  asignar a la posición *datos*[1][total de comunidades +1] la población total

  asignar a las posiciones de *datos*[2][0] a *datos*[diasSimulacion + 1][0] a las fechas en orden

  rellenar la tabla con los resultados correspondientes en cada celda a comunidad y fecha

- Selector getTabla() : JTable

5.11.10 Datos públicos: Ninguno.

5.11.11 Datos privados:

- tabla : JTable

5.12 Clase GraficoDatos

5.12.1 Tipo: Abstracción funcional

5.12.2 Objetivo: Obtener la representación de los datos en forma de gráficos.

5.12.3 Función: Configurar a partir de los resultados de una simulación, sendos gráficos que muestren la evolución de los datos totales y de los porcentajes para todas las comunidades y el total.

5.12.4 Subordinados: Ninguno.

5.12.5 Dependencias: Interfaz_Grafica.

5.12.6 Interfases:

- Utilizará los selectores de los objetos Resultado_Simulacion para crear los gráficos.

5.12.7 Recursos:

- Paquete jfree.chart para la creación de los gráficos.

5.12.8 Referencias: Ninguna.

5.12.9 Procesos:

- Constructor GraficoDatos(Resultado_Simulacion simulación)

  crear un gráfico de líneas XY JFreeChart *totales*

crear un gráfico de líneas XY JFreeChart *porcentajes*

crear ejes x e y de acuerdo a la poblacion total y diasSimulacion en *totales*

crear ejes x e y de 0 a 100 y diasSimulacion respectivamente en *porcentajes*

crear series para cada comunidad y total en *totales*

crear series para cada comunidad y total en *porcentajes*

asignar los valores correspondientes a comunidad, infectados y fecha en *totales*

asignar los valores correspondientes a comunidad, porcentaje y fecha en *porcentajes*

5.12.10 Datos públicos: Ninguno.

5.12.11 Datos privados:

- totales : JFreeChart
- porcentajes: JFreeChart

6. Matriz Requisitos-Componentes

Se presenta la matriz de correspondencia de requisitos con componentes:

	Clases											
Requisitos	Main	Almacen_Comunidades	Comunidades	Pais	Provincia	Pueblo	Almacen_Simulaciones	Resultado_Simulacion	Simulador	Interfaz_Grafica	TablaDatos	GraficoDatos
2.1.1.1	●	●	●	-	-	-	-	-	-	●	-	-
2.1.2	-	-	-	●	●	●	-	-	-	-	-	-
2.1.3.1	-	-	-	-	-	-	-	●	●	-	-	-
2.1.3.2	-	-	-	-	-	-	-	●	●	-	-	-
2.1.3.3	●	-	-	-	-	-	-	●	●	-	-	-
2.1.4.1	-	-	-	-	-	-	-	●	-	-	●	-
2.1.4.2	●	-	-	-	-	-	-	-	-	●	-	
2.1.5	●	●	-	-	-	-	-	-	●	●	-	-
2.1.6	-	●	●	-	-	-	-	-	-	-	-	-
2.1.7	-	-	-	-	-	-	●	●	-	●	-	-
2.1.8	-	-	-	-	-	-	-	-	-	-	-	●

Como se puede observar, todos los requisitos quedan cumplidos con los componentes diseñados.