Universidad Rey Juan Carlos

GRUPO 18 PRÁCTICA 1 EVOLUCIÓN Y ADAPTACIÓN DE SOFTWARE

Evolución y Adaptación del Software

Grado en Ingeniería del Software
Escuela Técnica Superior de Ingeniería Informática
Universidad Rey Juan Carlos

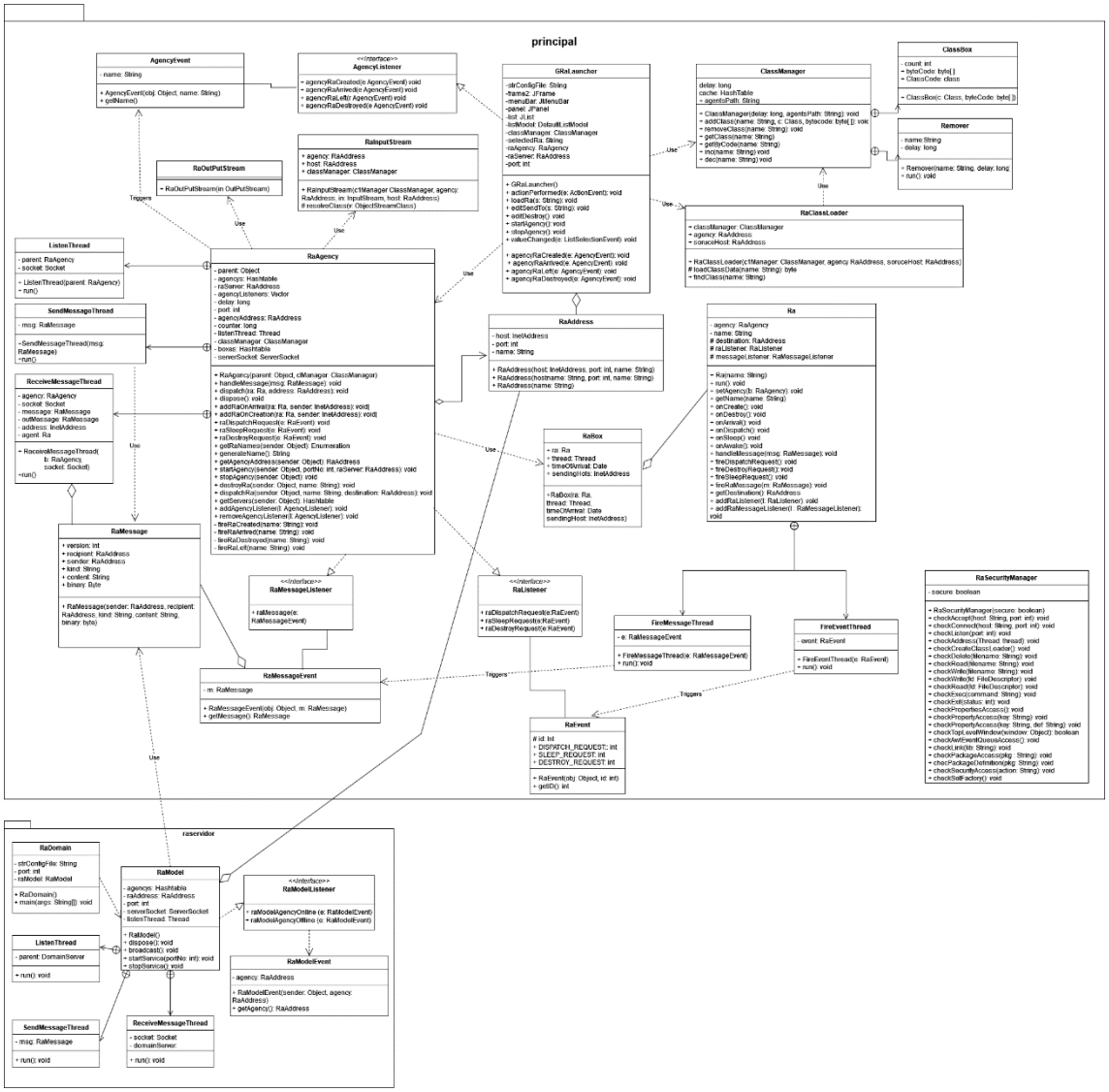
Autores

Stefano Tomasini Hoefner
Shuheng Ye
Gabriel Serrano Díaz
Jonathan Xavier Medina Salas

Instrucciones de ejecución

1. Recomendamos fuertemente usar *Intellij* como IDE
2. Cargar el proyecto, y poner como JDK openjdk-20 (en *File → Project Structure)*
3. En la carpeta *config*, fichero agenthandler.config, poner de *raDomainServerIp* la IP del ordenador que esté haciendo de servidor de dominio. Puede ser el actual si es el que está ejecutando *DomainServer.java*
4. En el mismo fichero, poner *copyClassFiles* a *true* para que una vez se compilen los agentes, se copien automáticamente sus .class a donde deben estar para que se vean dentro de la aplicación gráfica
5. Ejecutar *AgentHandler.java* una vez, y cerrarlo. Esto es solo para que se compilen los agentes.
6. Comentar los agentes en sus respectivos ficheros .java para evitar posibles conflictos con los agentes compilados que se van a cargar cuando se use la aplicación.
7. Una vez que se tienen los agentes compilados en su lugar, ejecutar la aplicación gráfica de manejo de agentes (*AgentHandler.java*).
8. Usar los agentes básicos, los que tienen clases internas o dependen de Java Swing causan muchos problemas.

Versión original del software



Fase de mantenimiento

Los primeros errores que fueron corregidos fueron los causados por actualizar el JDK del proyecto a JDK 20.0.1. Los siguientes que se arreglaron fueron los errores de red causados por que el código tenía algunas cosas desconfiguradas (puestas a nulo), lo cual requirió investigar bastante ver qué es lo que debería estar en cada campo o parámetro.

El código estaba bastante "desarreglado", así que se colocaron todas las tabulaciones bien, se renombraron varias variables para que dejen de ser de una sola letra, y se le dieron nombres más significativos y explicativos a clases, métodos y atributos.

Se le especificaron el tipo de los elementos de los vectores, enumerados, y otros tipos de estructuras de datos dentro de los "*< >*" característicos del sistema de genéricos de Java, para abandonar casi totalmente la utilización del *casting*. Ahora, en tiempo de compilación, el sistema de genéricos de Java se encarga de verificar si encajan los tipos de los elementos que componen las estructuras de datos cuando estas se traspasan de unas (*Hashtable<T, U>*) a otras (*Enumeration<T>*), en vez de hacerlo en tiempo de ejecución, que es menos fiable. Esto sería mantenimiento preventivo.

En *GraLauncher* (ahora llamado *AgentHandler*) se lanzaban excepciones de forma arbitraria solo porque el usuario no haya seleccionado a un agente antes de hacer click en el botón *edit*, o porque no haya seleccionado una agencia antes de presionar "enviar agente". Esto ya no sucede, ya que consideramos que eso no corresponde un uso apropiado de las excepciones, todo eso está comprendido dentro del funcionamiento normal de la aplicación, solo que no se cumplió un prerrequisito por parte del usuario antes de usar las respectivas funcionalidades, no es ningún fallo de aplicación. Se sustituyeron por el lanzado de mensajes de avisos dentro de la interfaz gráfica.

La selección de agentes también causaba algunos errores que se corrigieron (mantenimiento perfectivo).

Se reemplazó el uso de *System.out.println()* por la utilización de un *logger* en la mayoría de las clases, el cual es más adecuado para realizar *debugging*. Además, se mejoraron los mensajes escritos para que sean más informativos, y digan precisamente de qué parte del código surgen.

Se eliminó el uso de *Kaariboga* (librería de agentes móviles que ya no está disponible) para todos los agentes, a favor de utilizar los métodos equivalentes que ya están en RAF. De esta forma se consiguió salvar a todos los agentes que ya estaban, aunque todavía hay varios que no funcionan debido a sus otras dependencias que causan problemas al enviar el agente. Los que son básicos y no invocan librerías gráficas ni tienen clases internas funcionan.

Se borró la clase *RaDomain*, trae una complejidad excesiva al software solo para leer un único número de un fichero de propiedades con el que inicia *RaModel*, ahora se hace directamente en el ahora llamado *DomainServer*. Esto es mantenimiento preventivo estructural, aunque a muy pequeña escala.

Se simplificó la estructura sobre-complicada entre *RaAgency*, *ClassLoader* y *ClassManager* que había para cargar clases. Originalmente, para conseguir una clase, *ClassLoader* abría una conexión por un socket con la agencia y le mandaba un *RaMessage* de tipo *GET_CLASS* pidiéndole la clase especificada según su nombre, en el atributo *content* del mensaje. *RaAgency* se encargaba de conseguir el *bytecode* de la clase con ClassManager, este extrae los bytes del fichero .class del agente (si no estaba ya en su caché). Tras conseguir el bytecode, RaAgency le

respondía a *ClassLoader* con un *stream* que volcaba los bytes de la clase compilada. Finalmente, Classloader devolvía la instancia específica de *Class* que resultaba ser el agente compilado tras ser definido.

Ahora, se quitó *ClassLoader* enteramente, y únicamente hay una comunicación de pedir clase → devolver clase entre la *Agencia y ClassManager*, pero sin usar un socket. Además, se hizo que *ClassManager* sea solo una interfaz que es implementada por *CacheClassManager*. De esta forma, se reduce el acoplamiento entre la implementación específica que tenga el *ClassManager* específico y la agencia. Ahora, la clase concreta tiene que encargarse de cumplir correctamente su contrato con los métodos de la interfaz, y por su lado, la agencia no tiene que ajustarse a la implementación de esos métodos. Solo necesita poder: añadir agente, borrar agente (su *Class*), y conseguir la *Class* correspondiente a un *name*.

Se agregaron una gran cantidad de comentarios *Javadoc* sobre clases y métodos, para facilitar el entendimiento del código para aquellos desarrolladores que vayan a mantener alguna sección de código complicada, pero no se llegó a cubrir todo porque resultaba ser demasiado laborioso.

Se reemplazó el uso de un tipado basado en s*trings* para definir el tipo de *RaMessage* (antes), por el uso de un *enum* definido internamente en la clase, lo cual es más fiable que a la hora de hacer comprobaciones de valor del atributo *kind* de *RaMessage (ahora)* que haciéndolo con un *String.* Se hizo algo equivalente en la clase *AgentEvent* (antes llamada *RaEvent*), que anteriormente especificaba el tipo de evento con un entero de posible valor 1 hasta 3, pero ahora con un *enum*. Esto constituiría mantenimiento preventivo.

El énfasis principal que tuvo nuestro grupo en la fase de mantenimiento fue en hacer el código más auto-explicativo, y reducir la complejidad para que se pueda entender más fácil qué es lo que hace el programa. Obviamente, también se arreglaron los errores que había para poder alcanzar un estado funcional de la aplicación de agentes móviles, pero esto es algo que tienen que hacer necesariamente todos los grupos.

Propuestas de evolución

1. Quitar de la clase *GraLauncher* la responsabilidad de manejar la visualización, ceder esta responsabilidad a otra clase especializada para esto. De esta forma, se consigue desacoplar la lógica específica a la visualización, de la lógica que manipula a aquello que se va a visualizar.
2. Mejorar el manejo de la aplicación gráfica.
3. Quitar de la clase *AgentHandler* (antes *GraLauncher)* y *DomainServer* las responsabilidades de realizar el manejo de ficheros de propiedades para extraer las direcciones y puertos. Ceder esta responsabilidad a otra clase especializada para esto. De lo contrario, se inunda la clase original con código que solo entorpece el entendimiento de las funcionalidades esenciales de la clase.
4. Hacer que se copien los agentes compilados automáticamente a donde deberían estar para que aparezcan en la aplicación apenas se compilen (también facilita el *debugging*).
5. Crear algunos agentes nuevos.
6. Volver ClassManager un singleton.

Implementación de las propuestas

Propuestas 1 y 2:

AgentHandler.java (antes llamado *GRaLauncher*)

A esta clase se le quitó toda la lógica de definición de la interfaz gráfica de Swing. Ahora su constructor es mucho más corto y se entiende directamente qué es lo único que hace, lanzar la aplicación de escritorio y la agencia local.

AgentHandlerView.java

Esta nueva clase es la encargada de generar la interfaz gráfica. Aparte de que ahora la *GUI* tiene más funcionalidad (mensajes de avisos, y no permitir abrir el menú de acciones sobre el agente seleccionado si no se seleccionó ninguno), esta se comunica con AgentHandler mediante una interfaz:

AgentHandlerController.java

En realidad, es una clase abstracta en vez de una *interface*. Esto es porque si es una *interface* se me obliga a que todos los métodos sean públicos, pero yo quería que sean *default* para que estos métodos implementados por *AgentHandler* solo sean visibles dentro del paquete *agenthandler*.

Propuestas 3 y 4:

ConfigLoader.java

Para implementar estas propuestas, se creó esta clase, y a *AgentHandler* y *DomainServer* se le quitó todo el manejo de los ficheros *.properties* que leían para formar su configuración de red. Había una gran cantidad de bloques *try-catch* que solo ofuscaba las tareas más importantes que realizan las clases.

La nueva clase es ahora la encargada de realizar la extracción de la información de los ficheros *.properties*. Esto quita una gran cantidad de lógica de la clase *AgentHandler* en donde estaba, y reduce la carga cognitiva para los desarrolladores que tengan que mantenerla. Además, se volvio un *singleton*, para prevenir que haya más de una instancia.

Propuesta 5:

TicTacToe.java

Se creo esta clase con el fin de poder jugar ta-te-ti mediante un agente que guarde el estado del juego en sí. Aunque no se terminó de implementar, pareció ser una buena idea. No se pudo

terminar porque causaban problemas las dependencias con la librería Swing al enviar el agente compilado.

JumpCounter.java

Se creó esta clase básica solo con propósitos demostrativos de lo que hacen varios método base de Ra. Aunque al arreglar los agentes que ya existían, pero usaban Kaariboga, me di cuenta de que ya existía uno que ya hace exactamente lo mismo (*HelloAgent.java*)

RussianRoulette.java

Básicamente, cuando se crea el agente se decide un número entre 1 y 6, y si el nº de tu *dispatch* coincide con ese número, se te apaga la JVM. Es decir, se va manteniendo la cuenta de los *dispatches* realizados hasta el momento.

Propuesta 6

CacheClassManager.java

Ahora esta clase tiene un constructor privado, y una instancia estática final que viene pre-construida a nivel de clase. Esta única instancia se accede mediante el método estático *getInstance().*

Versión evolucionada del software

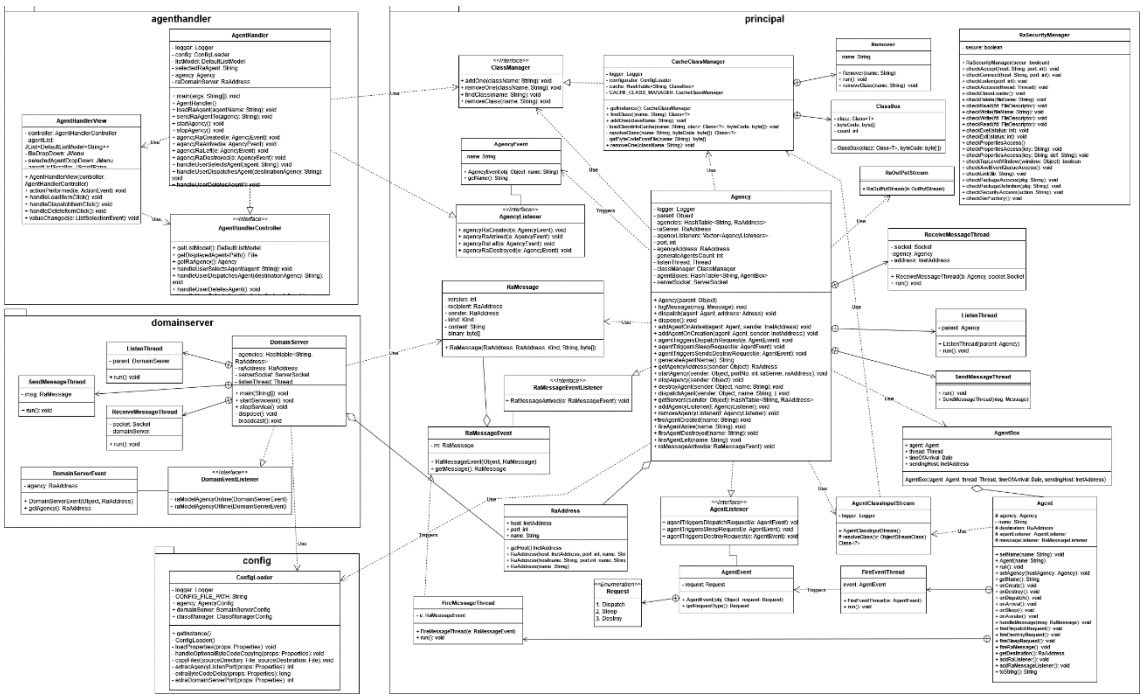


Figura 1. Diagrama UML del software evolucionado