Especificación de Diseño de Software

Santiago Videla

12 de octubre de 2010

Índice

1. Introducción **3**
1.1. Propósito . . . 3
1.2. Descripción general del documento . . . 3

2. Consideraciones de Diseño **3**
2.1. Objetivos . . . 3
2.2. Metodología . . . 3
2.3. Dependencias . . . 4
2.4. Herramientas y convenciones . . . 4

3. Arquitectura del Sistema **4**

4. Diseño de alto nivel **6**
4.1. Interfaces - Responsabilidades - Colaboradores . . . 6
4.1.1. IPluginAdmin . . . 6
4.1.2. IPlugin . . . 6
4.1.3. IFold . . . 6
4.1.4. IFoldInverse . . . 6
4.1.5. IStructureCmp . . . 8
4.1.6. ISequenceCmp . . . 8
4.1.7. ICombinatoryRegion . . . 8
4.1.8. ISolution . . . 8
4.1.9. INeighborhood . . . 8
4.1.10. IStrategy . . . 9
4.1.11. ICombinatoryEngine . . . 9
4.1.12. IQARegion . . . 9
4.1.13. IQAEngine . . . 9
4.1.14. IRanker . . . 9

5. Diseño de bajo nivel **11**
5.1. Paquetes y clases concretas . . . 11
5.1.1. Combinatory . . . 11
5.1.2. LocalSearch . . . 11
5.1.3. Region . . . 13
5.1.4. Validator . . . 15
5.1.5. LibRNA . . . 16
5.1.6. Ranker . . . 17
5.1.7. PluginAdmin . . . 17
5.1.8. Plugin . . . 18

1. Introducción

1.1. Propósito

El propósito de este documento es la especificación de diseño de software para la primer versión del producto "vac-o".

La confección de este documento se enmarca dentro del desarrollo de la tesis de grado de la carrera Lic. en Cs. de la Computación de la FaMAF - UNC, **"Diseño de vacunas atenuadas con menor probabilidad de sufrir reversión a la virulencia"** a cargo de Santiago Videla, con la dirección de la Dra. Laura Alonso i Alemany (FaMAF) y la colaboración de Daniel Gutson (FuDePAN).

El documento esta dirigido a las personas involucradas en el desarrollo de la tesis como así también a los colaboradores de FuDePAN que eventualmente podrían participar en las etapas de desarrollo y mantenimiento del software.

1.2. Descripción general del documento

En la sección 2 se mencionan los objetivos, la metodología adoptada y las dependencias del diseño.

En la sección 3 se muestra la arquitectura del sistema con sus principales componentes e interacciones.

En la sección 4 se presenta el diseño de alto nivel del sistema, sus interfaces y paquetes principales.

En la sección 5 se presenta el diseño de bajo nivel del sistema, las clases concretas y sus relaciones para cada paquete.

2. Consideraciones de Diseño

2.1. Objetivos

Principalmente se pretende lograr un diseño que cumpla con los principios fundamentales del diseño orientado a objetos, comúnmente conocidos por el acrónimo "SOLID" [1].

En particular, se pone especial énfasis en respetar los principios OCP (*Open-Closed Principle*) y DIP (*Dependency Inversion Principle*) debido a su importancia para obtener un sistema que sea fácilmente extensible y configurable con el fin de satisfacer las necesidades de los usuarios.

2.2. Metodología

La metodología utilizada para realizar el análisis y descripción del diseño se denomina "Diseño dirigido por responsabilidades" [2]. Esta técnica se enfoca en *qué* acciones (responsabilidades) deben ser cubiertas por el sistema y que objetos serán los responsables de llevarlas a cabo. *Cómo* se realizara cada acción, queda en un segunda plano.

2.3. Dependencias

Se asume para la confección de este diseño, el acceso a diferentes librerías externas que serán fundamentales para el correcto funcionamiento del sistema en su conjunto. Respetando la metodología adoptada, no se hará referencia directa a una u otra librería, sino a los servicios que las mismas debe ser capaces de proveer al sistema.

2.4. Herramientas y convenciones

Se utiliza UML[3] como lenguaje de modelado y ArgoUML[4] como herramienta para la confección de diagramas. Además se adopta la convención de nombrar a las interfaces anteponiendo una *"I"* al nombre de la clase concreta que la implementa (interface: *"IPersona"*, clase concreta: *"Persona"*).

3. Arquitectura del Sistema

En la Figura 1 se presenta un diagrama de la arquitectura del sistema y la interacción entre sus componentes principales. A continuación se da una breve descripción de cada uno de ellos:

- **Main:** Representa el componente principal en términos de ejecución del sistema. Comprende principalmente, la inicialización y configuración de otros componentes.
- **Plugin:** Representa la implementación de las características propias de la vacuna que se desea optimizar. Brinda información de configuración inicial como así también, los criterios para evaluar una secuencia.
- **CombinatoryEngine:** Representa el motor combinatorio del sistema encargado de encontrar, nuevas secuencias que sean candidatas a optimizar la atenuación de la vacuna.
- **QAEngine:** Representa el motor de control de calidad del sistema encargado de decidir, si una secuencia candidata pasa o no dicho control.
- **Ranker:** Representa el componente encargado de mantener un "ranking" de secuencias.
- **libRNA:** Representa el componente que provee al sistema funcionalidades para la manipulación de secuencias de ARN ("folding" directo e inverso) utilizando librerías externas (Vienna Package, UNAFold, entre otras)

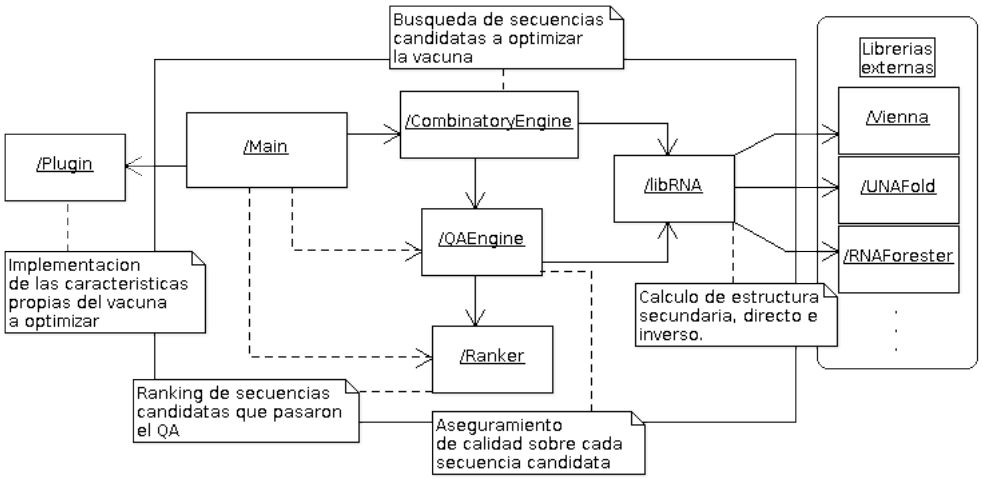

Figura 1: UML - Arquitectura

4. Diseño de alto nivel

4.1. Interfaces - Responsabilidades - Colaboradores

En esta sección se presentan las principales interfaces que intervienen en el sistema, sus respectivas responsabilidades y colaboradores. En la Figura 2 se puede ver el diagrama de clases correspondiente.

Finalmente, en la Figura 3 se presenta el diagrama de secuencia correspondiente a la comunicación entre las principales entidades del sistema.

4.1.1. IPluginAdmin

Responsabilidad: Administrar las extensiones del sistema (archivos *.so*).

1. Cargar extensión.

4.1.2. IPlugin

Responsabilidad: Brindar la información y servicios particulares para una vacuna determinada.

1. Proveer la lista de parámetros requeridos por la extensión.
2. Proveer la solución inicial conteniendo la secuencia de ARN que se encuentra en la cepa vacunal.
3. Proveer las regiones combinatorias que se deben usar para buscar mejoras a la vacuna.
4. Proveer el umbral que se debe usar para determinar la bondad de las secuencias obtenidas de las regiónes combinatorias.
5. Proveer las regiones de validación que se deben usar para realizar el control de calidad.
6. Proveer el vecindario para la búsqueda local.
7. Proveer la estrategia de búsqueda local.
8. Determinar si se continua buscando secuencias o no.
9. Evaluar las soluciones candidatas.
10. Descargar la extensión.

4.1.3. IFold

Responsabilidad: Proveer al sistema el “folding” directo de secuencias ARN

Colaboradores:

1. Vienna Package, o UNAFold, u otros.

4.1.4. IFoldInverse

Responsabilidad: Proveer al sistema el “folding” inverso de secuencias ARN

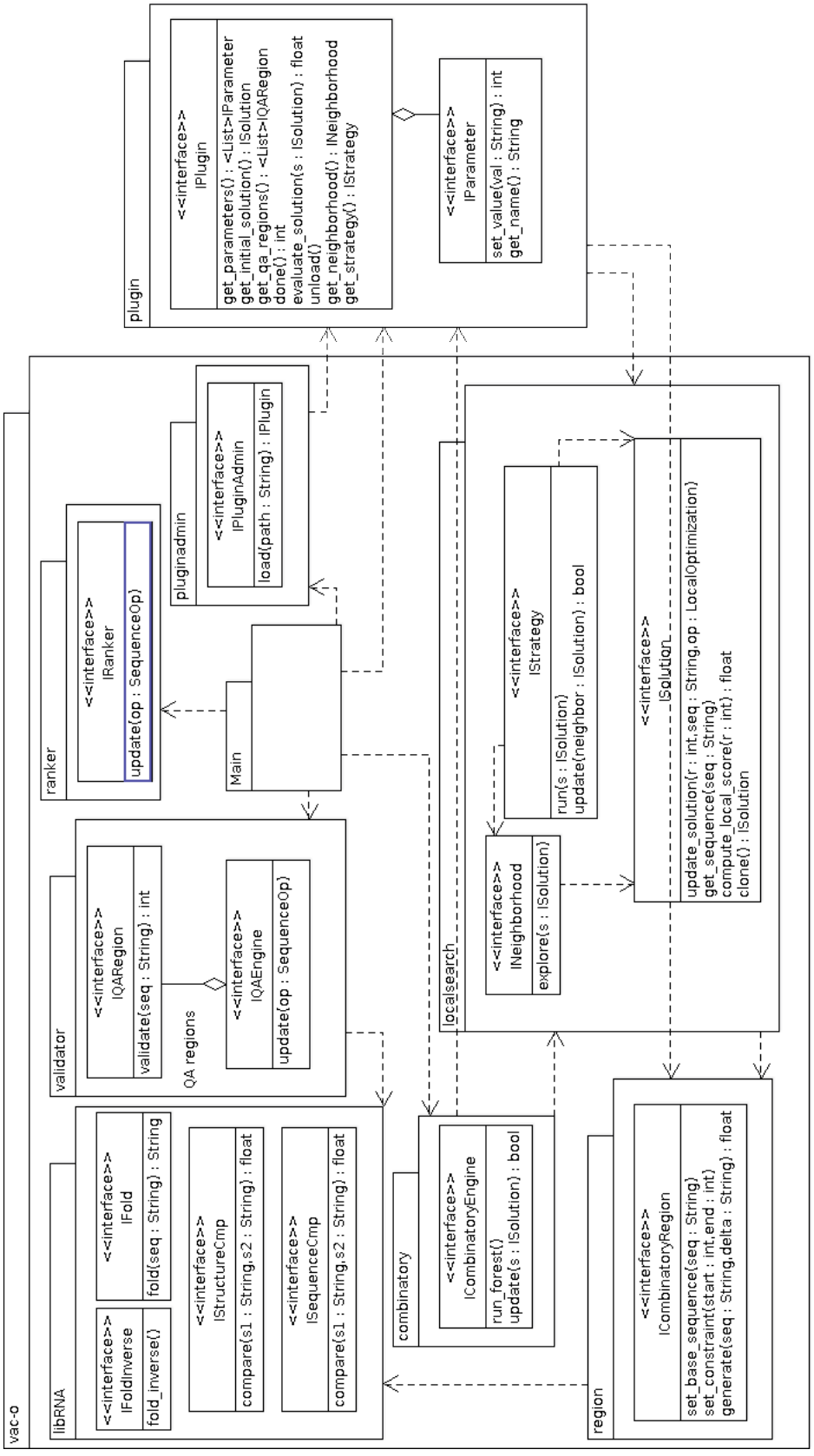

Figura 2: UML - Interfaces

Colaboradores:

1. Vienna Package u otros.

4.1.5. IStructureCmp

Responsabilidad: Proveer al sistema la comparación de estructuras secundarias.

Colaboradores:

1. RNAForester u otros.

4.1.6. ISequenceCmp

Responsabilidad: Proveer al sistema la comparación de secuencias de ARN.

Colaboradores:

1. Vienna Package u otros.

4.1.7. ICombinatoryRegion

Responsabilidad: Calcular las secuencias que mantengan determinadas propiedades de una secuencia original.

1. Devolver la siguiente secuencia, el cambio realizado y la evaluación local del cambio.

Colaboradores:

1. IFold, IFoldInverse, IStructureCmp, ISequenceCmp

4.1.8. ISolution

Responsabilidad: Representar una solución candidata en el espacio de busqueda.

1. Proveer la secuencia completa de la solución.
2. Actualizar una componente (región) de la solución y la secuencia completa resultante.
3. Calcular la evaluación local de la solución como producto de la evaluación de sus componentes (regiones).

4.1.9. INeighborhood

Responsabilidad: Explorar el vecindario de una solución.

Colaboradores:

1. **ICombinatoryRegion:** Consulta la siguiente secuencia de cada región combinatoria.

4.1.10. IStrategy

Responsabilidad: Establecer la política para pasar de una solución a la siguiente y notificar cada solución que mejora a las anteriores.

1. Seleccionar una solución entre los vecinos de la solución actual.

Colaboradores

1. **INeighborhood:** Explora el vecindario para cada solución.

4.1.11. ICombinatoryEngine

Responsabilidad: Generar secuencias candidatas a partir de las soluciones encontradas por la búsqueda local.

1. Iniciar la búsqueda local.
2. Notificar nuevas secuencias candidatas.

Colaboradores:

1. **IStrategy:** Ejecuta la búsqueda local.

4.1.12. IQARegion

Responsabilidad: Realizar el control de calidad para una región de validación.

1. Calcular y validar mutaciones acumuladas de la región hasta alcanzar la profundidad deseada.

Colaboradores:

1. IFold

4.1.13. IQAEngine

Responsabilidad: Realizar el control de calidad para una secuencia candidata.

1. Determinar si una secuencia candidata aprueba o no el control de calidad para todas sus regiones de validación.

Colaboradores:

1. **IQARegion:** Consulta si la región de validación aprueba o no el control de calidad.

4.1.14. IRanker

Responsabilidad: Mantener un *ranking* de secuencias en base a evaluación global de cada una de ellas.

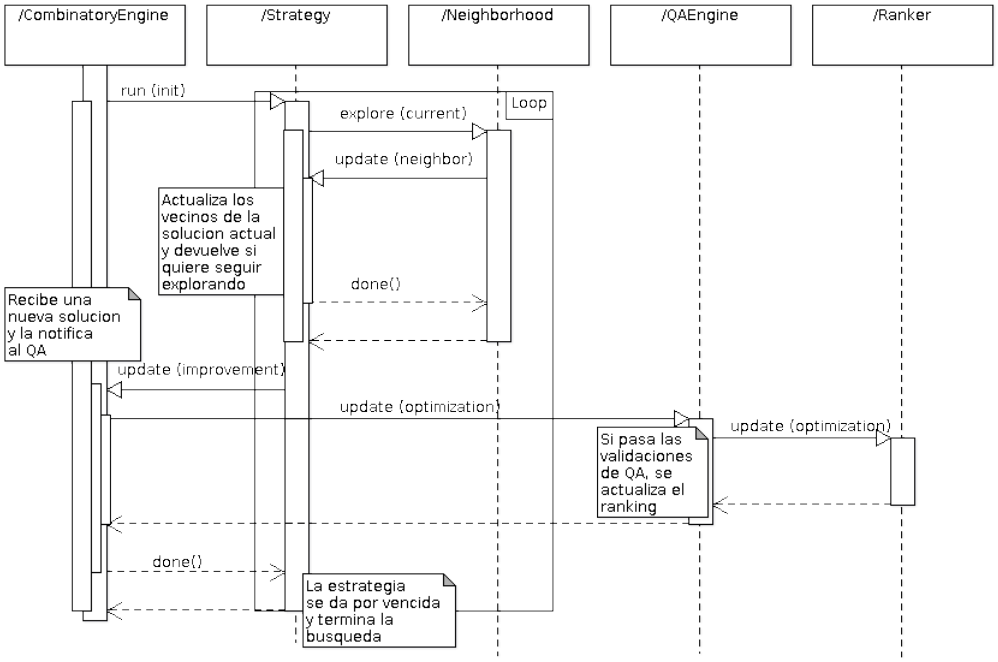

Figura 3: UML - Pasaje de mensajes

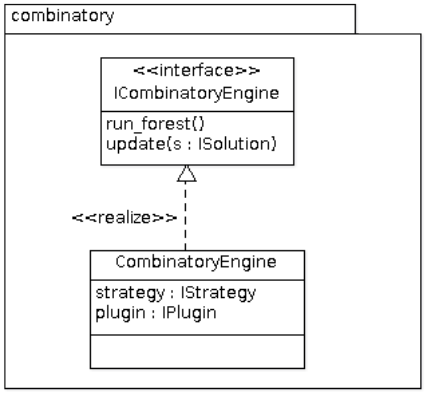

Figura 4: UML - Combinatory

5. Diseño de bajo nivel

5.1. Paquetes y clases concretas

En esta sección se presentan las clases concretas que implementan las interfaces presentadas en la sección 4. Para mayor claridad, se dividieron los diagramas UML por paquetes.

5.1.1. Combinatory

En la Figura 4 se puede ver el diagrama de clases para el paquete *combinatory*. Este paquete forma parte del componente "CombinatoryEngine" en la Figura 1 de la sección 3.

La responsabilidad de este paquete es brindar al sistema, la interfaz al motor combinatorio ocultando la implementación de la estrategia de búsqueda utilizada.

5.1.2. LocalSearch

En la Figura 5 se puede ver el diagrama de clases para el paquete *localsearch*. Este paquete forma parte del componente "CombinatoryEngine" en la Figura 1 de la sección 3.

La responsabilidad de este paquete es brindar al motor combinatorio, la implementación de diferentes estrategias de búsqueda local para la optimización de una solución inicial (la vacuna a optimizar). Inicialmente, se consideran la "familia" de algoritmos de búsqueda local por "mejoramiento iterativo". Algunos de estos algoritmos son:

- HillClimbing (FirstImprovement o BestImprovement)
- Simulated Annealing

Cada una de estas estrategias difieren una de las otras, en como se produce el paso de una solución a la siguiente. La interface "IStrategy" define los métodos que deben implementar las clases para cada estrategia.

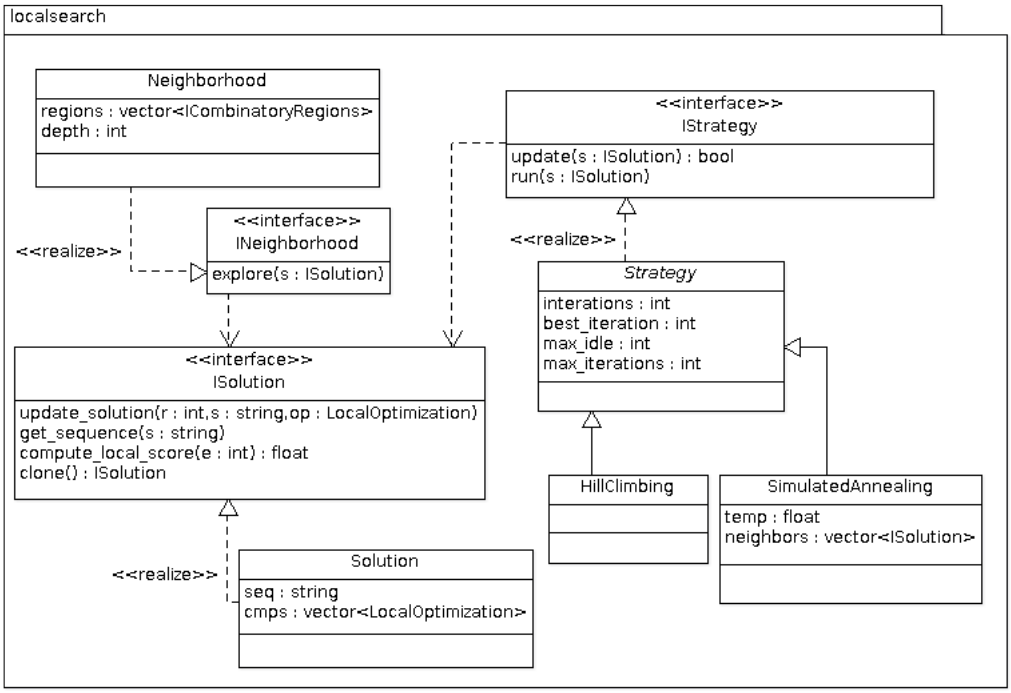

Figura 5: UML - LocalSearch

Otro elemento fundamental para la búsqueda local, es la definición de lo que se denomina el vecindario de cada solución. Es decir, una relación entre soluciones del problema, que permita ir de una solución a la siguiente. Esto esta representado por la interface "INeighborhood". Cada clase concreta que la implemente, define un tipo diferente de vecindario. Un ejemplo concreto de vecindario puede ser el siguiente:

Sea S el conjunto de todas las soluciones (secuencias de ARN que satisfacen las restricciones impuestas), definimos para $0 < i$, $N(i) \subseteq S \times S$ tal que, el par $(s, s') \in S \times S$, pertenece a $N(i)$ si y solo si, ocurre lo siguiente:

1. s' difiere de s en a los sumo una región
2. Si s' difiere de s en la región k, entonces se necesitaron a lo sumo i pasos para llegar de s_k a s'_k
3. Sean k_i las regiones de s', entonces se cumple $\prod_i score(k_i) \geq cutoff$

Donde $score$ es la función que devuelve la evaluación local de una región y $cutoff \in (0, 1)$ es el umbral de aceptación.

Las diferentes combinaciones entre definiciones de vecindarios y estrategias impactaran en los resultados de la búsqueda.

5.1.3. Region

En la Figura 6 se puede ver el diagrama de clases para el paquete *region*. Este paquete forma parte del componente "CombinatoryEngine" en la Figura 1 de la sección 3.

La principal responsabilidad de este paquete es brindar al sistema, la implementación de diferentes regiones (restricciones) combinatorias sobre secuencias de ARN. Para la primer versión de "vac-o" y según lo establece la especificación de requerimientos, se contemplan dos tipos de restricciones.

- Conservación de la estructura secundaria (*SSRegion*).
- Conservación del código genético (*GCRegion*).

El segundo caso, no merece mayor detalle a nivel de diseño. Por otro lado, para la restricción de conservar la estructura secundaria, vale la pena profundizar en que implica garantizar esta responsabilidad.

La clase *SSRegion* genera aquellas secuencias de ARN que conservan una estructura secundaria dada (*vaccine_structure*). Ya que la cantidad de secuencias que conservan una misma estructura secundaria podría ser eventualmente muy grande, se ofrecen dos restricciones complementarias para reducir el numero de secuencias que son tenidas en cuenta. A continuación resumimos brevemente cada una de estas restricciones:

- **Distancia mínima** a las secuencias que conservan una estructura secundaria dada (*wt_structure*). Aquellas secuencias que estén a menor distancia que la distancia mínima, serán descartadas. La cantidad de secuencias con las que se compara y calcula la distancia, lo determina el valor de *wt_seq_cache*.

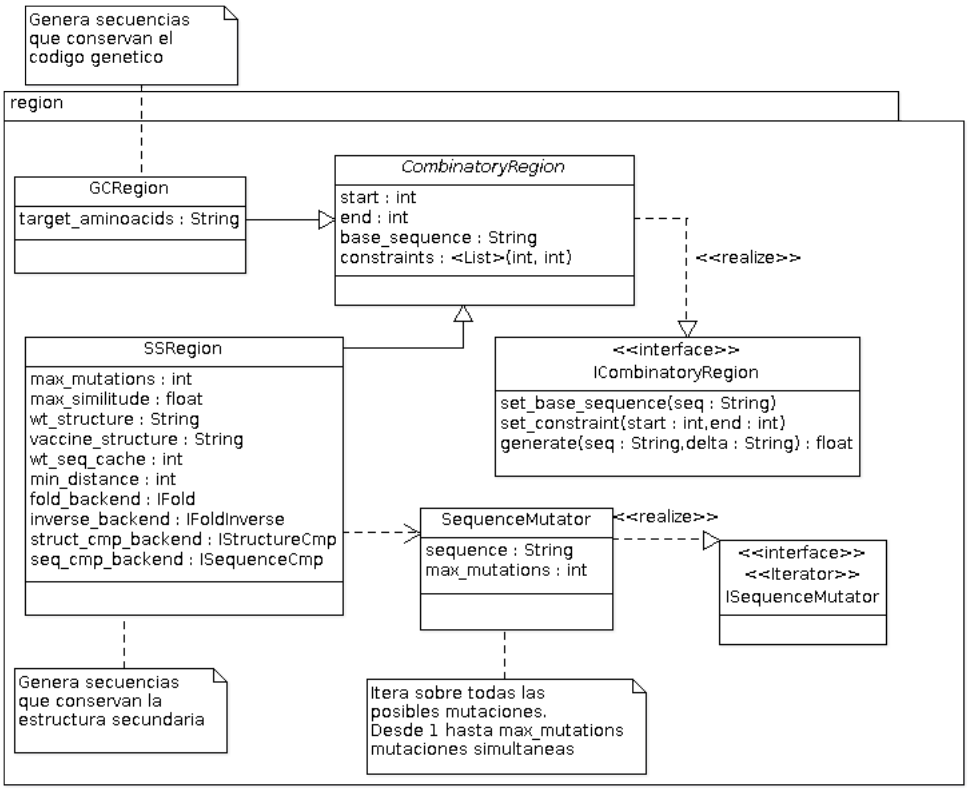

Figura 6: UML - Region

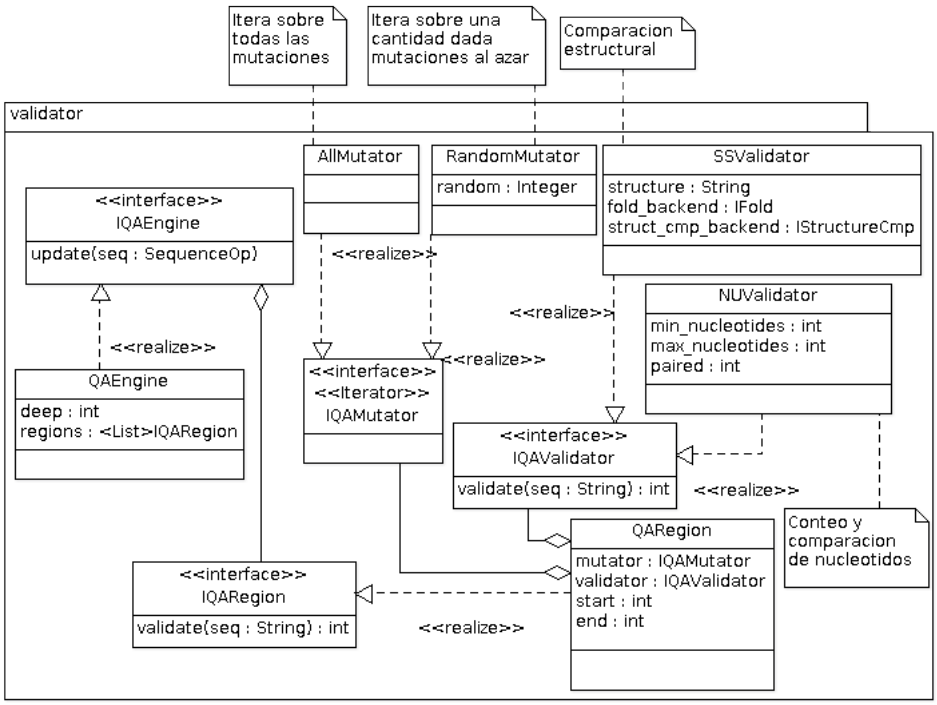

Figura 7: UML - Validator

- **Similitud estructural** a una estructura secundaria dada (*wt_structure*). Esta restricción implica un mayor costo computacional ya que partiendo de una posible secuencia, se realizan desde una, hasta *max_mutations* mutaciones simultaneas sobre toda la secuencia, y para cada posible mutación, se comprara la estructura secundaria de la mutación con la estructura *wt_structure*. Aquellas secuencias que excedan el porcentaje *max_similitude* de similitud, serán descartadas.

5.1.4. Validator

En la Figura 7 se puede ver el diagrama de clases para el paquete *validator*. Este paquete representa el componente "QAEngine" en la Figura 1 de la sección 3.

La responsabilidad de este paquete es realizar una serie de pruebas que garanticen al sistema que una secuencia dada, merece ser evaluada y tenida en cuenta como posible optimización de la vacuna. Las pruebas que se realizan para garantizar la "calidad", se basan en que luego de sufrir alguna cantidad acumulada de mutaciones, la secuencia mantenga determinadas propiedades en determinadas regiones.

En la primer versión de "vac-o" se contemplan dos maneras de generar mutaciones de una secuencia:

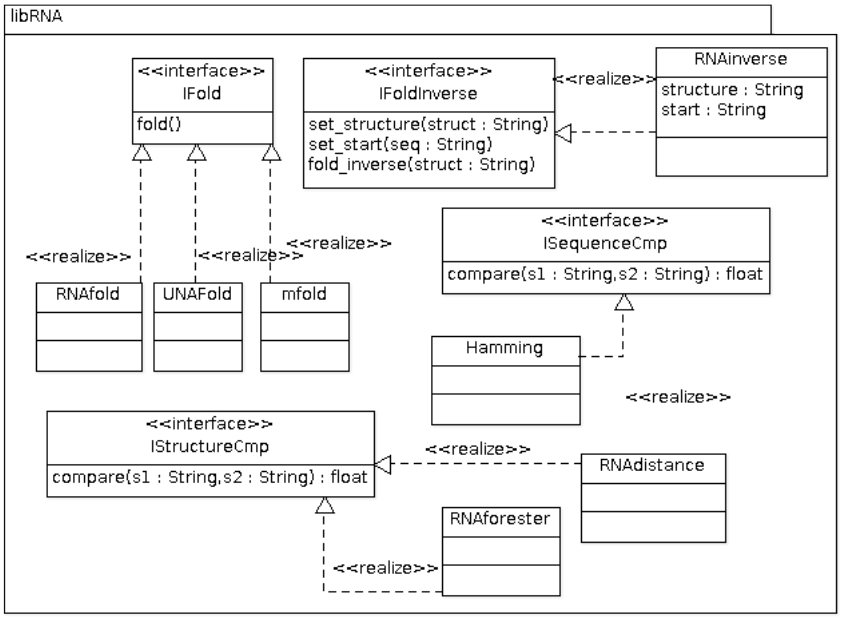

Figura 8: UML - LibRNA

- Todas las mutaciones posibles (*AllMutator*)
- Una cantidad (*random*) de mutaciones al azar (*RandomMutator*)

y dos propiedades a verificar sobre cada mutación:

- Comparación estructural con una estructura secundaria dada (*SSValidator*).
- Conteo y comparación de nucleótidos apareados o desapareados (*NUValidator*).

5.1.5. LibRNA

En la Figura 8 se puede ver el diagrama de clases para el paquete *libRNA*. Este paquete representa el componente "libRNA" en la Figura 1 de la sección 3.

La responsabilidad de este paquete es brindar al sistema servicios para la manipulación de secuencias de ARN. Para cumplir con esta responsabilidad, se ofrece al sistema el acceso a librerías externas de manera transparente y permitiendo utilizar diferentes librerías para acceder a diferentes servicios.

En la primer versión del sistema, se contemplan los siguientes servicios como indispensables, aunque en futuras versiones se podrían agregar otros:

- "Folding" directo (*IFold*)
- "Folding" inverso (*IFoldInverse*)

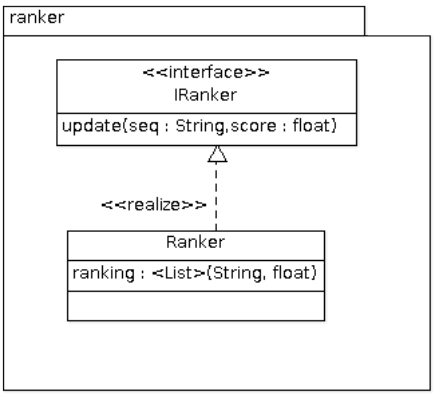

Figura 9: UML - Ranker

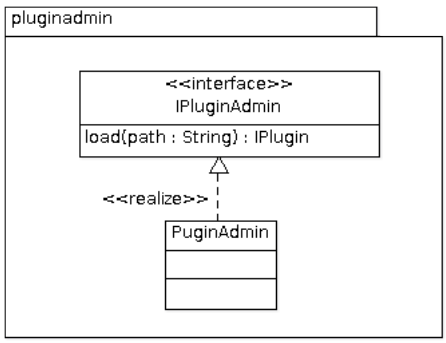

Figura 10: UML - PluginAdmin

- Comparación entre estructuras (*IStructureCmp*)
- Comparación entre secuencias (*ISequenceCmp*)

La importancia de este paquete y las interfaces que contiene radica en que le permite al resto del sistema, abstraerse del uso de una u otra librería, y contar con una API unificada para acceder a estos servicios.

5.1.6. Ranker

En la Figura 9 se puede ver el diagrama de clases para el paquete *ranker*. Este paquete representa el componente "Ranker" en la Figura 1 de la sección 3.

La responsabilidad de este paquete es mantener el *ranking* de secuencias candidatas a optimizar la vacuna.

5.1.7. PluginAdmin

En la Figura 10 se puede ver el diagrama de clases para el paquete *pluginadmin*. Este paquete no aparece explícitamente en la arquitectura del sistema

(Figura 1 de la sección 3), pero lo podemos identificar con la flecha que une el componente “Main” con “Plugin”.

Fundamentalmente la responsabilidad de este paquete, es brindar al sistema la funcionalidad de cargar las extensiones en memoria.

5.1.8. Plugin

Para el paquete *plugin* no se especifica un diseño de bajo nivel debido a que la implementación de cada extensión no forma parte del sistema “vac-o”. Simplemente, se asume que las extensiones que se implementen en el futuro, deberán garantizar que cumplen con el “contrato” establecido por las interfaces de este paquete.

Referencias

[1] Design Principles and Design Patterns. Robert C. Martin, 2000. `www.objectmentor.com`

[2] Rebecca Wirfs-Brock and Alan McKean. Object Design: Roles, Responsibilities and Collaborations, Addison-Wesley, 2003

[3] Unified Modeling Language: `http://www.uml.org/`

[4] ArgoUML: `http://argouml.tigris.org/`