DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico de Diseño y Desarrollo, Huerta orgánica de precisión

Ingeniería del Software 2

Integrante	LU	Correo electrónico
Guillermo Gallardo Diez	032/10	gagdiez@hotmail.com
Manuel Ferreria	199/10	mferreria@gmail.com
Luciano Gandini	207/10	gl.gandini@gmail.com
Luis Scoccola	382/10	luis.scoccola@gmail.com
Fabrizio Borghini	406/10	fabriborghini@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Contents

1 *Product Backlog*

A continuación se encuentran las *stories* que logramos condensar al iniciar el trabajo práctico, a partir del enunciado.

Product Backlog		
Code	User Story	Points
INT1	**As a Gardener I Want to** check the PH, humidity and temperature of the ground through the Arduino sensor **So I Can** verify the actual plant conditions.	8
INT2	**As a Gardener I Want to** see get weather for tomorrow from the metheorological center **So I Can** see what actions will be taken.	5
INT3	**As a Gardener I Want to** input specific charateristics of the plant **So I Can** track these across time	3
INT5	**As a Gardener I Want to** check the historical characteristics of the plant **So I Can** track its growth.	2
INT4	**As a Gardener I Want to** check when was the last time actuators were activated **So I Can** call for maintenance if necessary.	1
INT6	**As a Gardener I Want to** get an estimate of when to harvest **So I Can** plan my meals.	13
INT7	**As a Gardener I Want to** view every action ever taken **So I Can** check for malfunctions.	2
INT8	**As a Gardener I Want to** receive a message on my phone when the cultivation plan changes **So I Can** be aware of any anomalies.	8
INT9	**As a Gardener I Want to** check the growth plan and actual status **So I Can** modify it if its wrong.	5
INT11	**As a Gardener I Want to** check the plan for the next 24 hours. **So I Can** schedule more actions.	2
INT10	**As a Gardener I Want to** automatize the actions to take **So I Can** follow the master plan.	8
INT12	**As a Gardener I Want to** automatize the actuators activation **So I Can** so actions can be taken automatically.	5
GAL1	**As a Botanist I Want to**specify care rules **So I Can** get warnings when the growth conditions are not ideal.	3
GAL2	**As a Botanist I Want to**input a master growth plan **So I Can** specify growth conditions for the plant.	3
GAL3	**As a Gardener I Want to** visualize the historical values of indicators and supplies. **So I Can** decide if the values are correct.	3
WEB1	**As a Gardener I Want to** check via web the status **So I Can** monitor my plant from around the world.	8

2 *Sprint Backlog*

Esta es la especificación de las *stories* que se incluyeron en el *sprint*. Notar que las tareas fueron refinadas la primera semana que comenzó el *sprint*. Básicamente, al comenzar a diseñar, notamos que las tareas habían quedado demasiado amplias, ya que no teníamos experiencia en programar un *sprint*. Esto se explica en secciones posteriores.

Sprint Backlog						
Code	User Story	Acceptance Criteria	Value	Points	Tasks	Description
INT1	**As a Gardener I Want to** check the PH, humidity and temperature of the ground through the Ardruino sensor **So I Can** verify the actual plant conditions.	(1.) The values displayed must be the ones measured by the sensors at the moment they're required.	7	8	(1.) simulate the information measured by the Arduino. (2.) retrieve the information measured from the Arduino interface. (3.) interpret the information correctly. (4.) display the information in a human-readable way.	Interface between the three sensors and the application. Interpret the information, save it (to make decisions) and display it so the user can read it.
INT2	**As a Gardener I Want to** get weather for tomorrow from the metheorological center **So I Can** see what actions will be taken.	(1.) The values displayed must be the ones forecasted by the central at the moment they're required.	5	5	(1.) simulate the information from the center. (2.) retrieve the information from the meteorological center. (3.) interpret the information correctly. (4.) display the information in a human-readable way.	Interface between the meteorological attachment and the application. Interpret the information, store it (to make future decisions), and display it so the user can read it.
INT10	**As a Gardener I Want to** automatize the actions to take **So I Can** follow the master plan	(1.) The system must automatically check the sensors value. (2.) The system must check with the master plan to check whether any actions have to be taken.	7	8	(1.) (INT1). (2.) (GAL2). (3.) decide what actions to take.	The application must have enough logic to decide the next action to follow based on the information retreived from the sensors.
INT12	**As a Gardener I Want to** automatize the actuators activation **So I Can** so actions can be taken automatically	(1.) The system must trigger the actuators if any actions have to be taken.	10	5	(1.) simulate the actuators incidence and response. (2.) (INT10). (3.) control the actuators.	The decitions made int (INT10) must be followed, by activating the actuators acordingly.
GAL2	**As a Botanist I Want to**input a master growth plan **So I Can** specify growth conditions for the plant	(1.) The system must have a valid way to input the values for every growth stage.	5	3	(1.) desing an interface so the user can input a master plan.	Interface that allows the botanist especify values for the PH, humidity and temperature of the ground for every growth stage.

3 Seguimiento

3.1 *Sprint Backlog* ajustado

Comenzamos refinando las tareas del *Product Backlog* inicial[1].

[1]Notar que cuando aparecen n horas asignadas a más de una persona, quiere decir que estas personas estuvieron n horas **en total** realizando esta tarea; no que que cada una estuvo n horas.

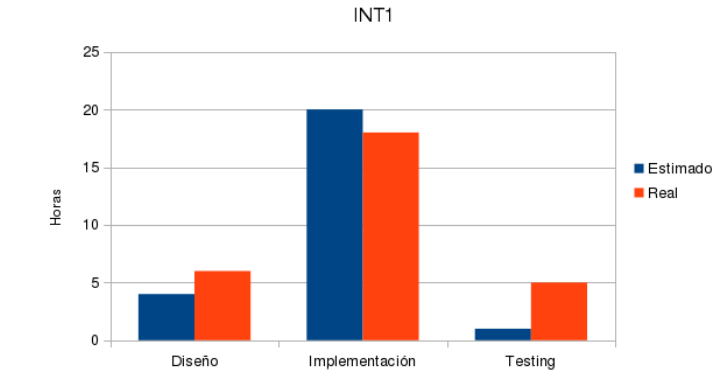
Sprint Backlog w/ elaborated tasks				
Code	User Story	Tasks	Estimated hours	Assigned to and used hours
INT1	**As a Gardener I Want to** check the PH, humidity and temperature of the ground through the Ardruino sensor **So I Can** verify the actual plant conditions.	**Design** 1. abstract the *Arduinos*. 2. design the sensors. 3. design the sensor manager. 4. design a class or classes for a types that represent the measurments. 5. design an atomatic way for the sensors to sensate and save the information. **Implementation** 1. retrieve the information measured from the Arduino interface. 2. interpret the information correctly. 3. display the information in a human-readable way. **Testing** 1. simulate the information measured by the Arduino.	(1.) **Design** $4hs$ (2.) **Implementation** $20hs$ (3.) **Testing** $1hs$	**Design** 1. 1 hora - Luis 2. 1 hora - Luis 3. 1 hora - Luis 4. 1 hora - Fabricio 5. 2 horas - Guillermo **Implementation** 1. 7 horas - Manuel 2. 7 horas - Manuel 3. 4 horas - Luciano y Guillermo **Testing** 1. 5 horas - Luciano y Manuel
INT2	**As a Gardener I Want to** get weather for tomorrow from the metheorological center **So I Can** see what actions will be taken.	**Design** 1. design the metheorological center. 2. design a class for a type that represents the measurements. 3. design an atomatic way for the metheorological center to sensate and save the information. **Implementation** 1. retrieve the information from the meteorological center. 2. interpret the information correctly. 3. display the information in a human-readable way. **Testing** 1. simulate the information from the center.	(1.) **Design** $2hs$ (2.) **Implementation** $4hs$ (3.) **Testing** $4hs$	**Design** 1. 1 hora - Guillermo y Luis 2. 1 hora - Fabricio 3. 1 hora - Guillermo **Implementation** 1. 2 horas - Manuel 2. 1 horas - Manuel 3. 2 horas - Luciano **Testing** 1. 5 horas - Luciano

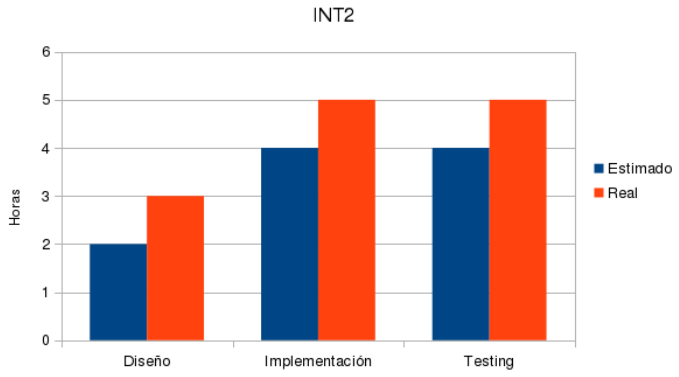
Sprint Backlog w/ elaborated tasks (cont.)				
Code	User Story	Tasks	Estimated hours	Assigned to and used hours
INT10	**As a Gardener I Want to** automatize the actions to take **So I Can** follow the master plan	**Design** 1. design a decision-maker. 2. design a class for the type of the decisions. **Implementation** 1. (INT1). 2. (GAL2). 3. decide what actions to take.	(1.) **Design** $4hs$ (2.) **Implementation** $20hs$ (3.) **Testing** $4hs$	**Design** 1. 6 horas - Luis y Guillermo 2. 1 hora - Luis y Guillermo **Implementation** 1. 20 horas - Manuel y Luciano **Testing** 1. 4 horas - Manuel
INT12	**As a Gardener I Want to** automatize the actuators activation **So I Can** so actions can be taken automatically	**Design** 1. design the actuators. 2. design an actuator manager. **Implementation** 1. (INT10). 2. control the actuators. 1. simulate the actuators incidence and response.	(1.) **Design** $2hs$ (2.) **Implementation** $4hs$ (3.) **Testing** $4hs$	**Design** 1. 1 hora - Guillermo y Luis 2. 1 hora - Guillermo y Fabricio **Implementation** 1. 5 horas - Luciano **Testing** 1. 6 horas - Manuel
GAL2	**As a Botanist I Want to**input a master growth plan **So I Can** specify growth conditions for the plant	**Design** 1. design an interface so the user can input a master plan. 2. design the master plan format. **Implementation** 1. implement the master plan in a persistent and yet modifiable fashion.	(1.) **Design** $2hs$ (2.) **Implementation** $4hs$ (3.) **Testing** $4hs$	**Design** 1. 4 horas - Guillermo y Luis 2. 1 hora - Fabricio **Implementation** 1. 8 horas - Manuel y Luciano **Testing** 1. 2 horas - Manuel y Luciano

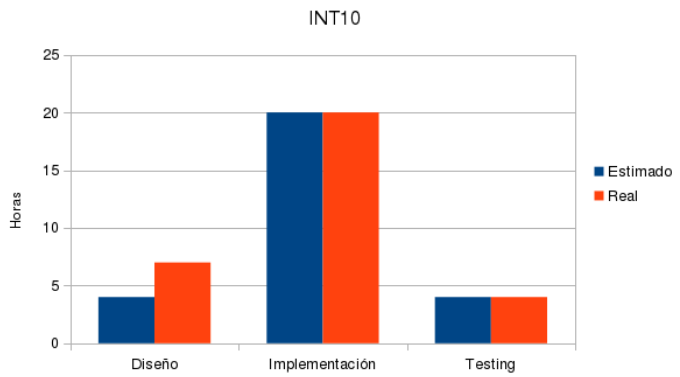
3.2 Burndown Charts

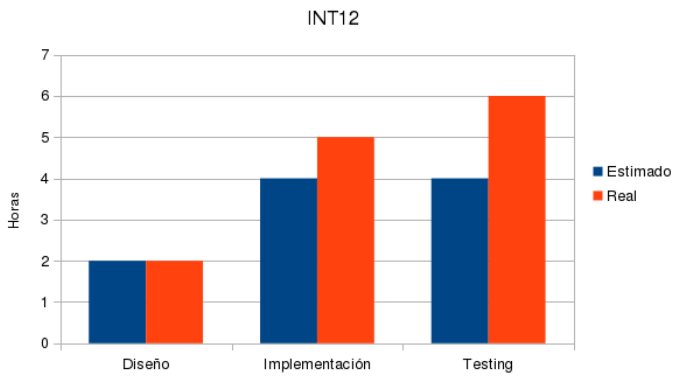
3.2.1 Comparación horas estimadas–horas reales

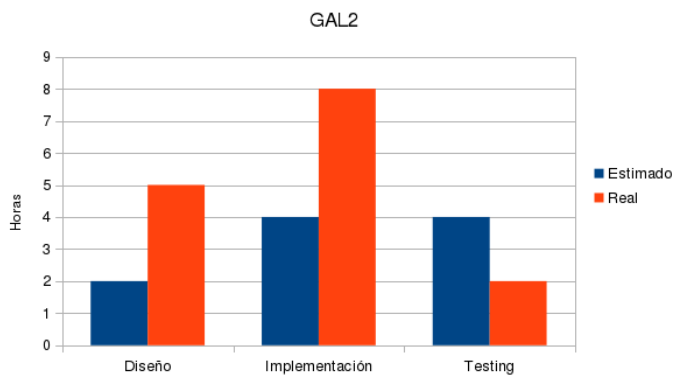
A continuación se encuentra la comparación entre horas estimadas y horas reales para cada una de las *stories* y para el *sprint* en general.

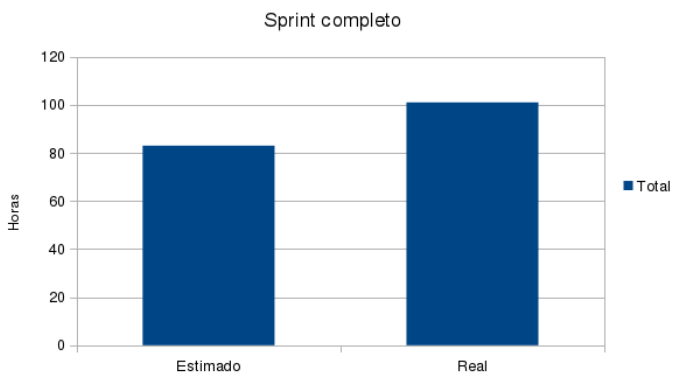






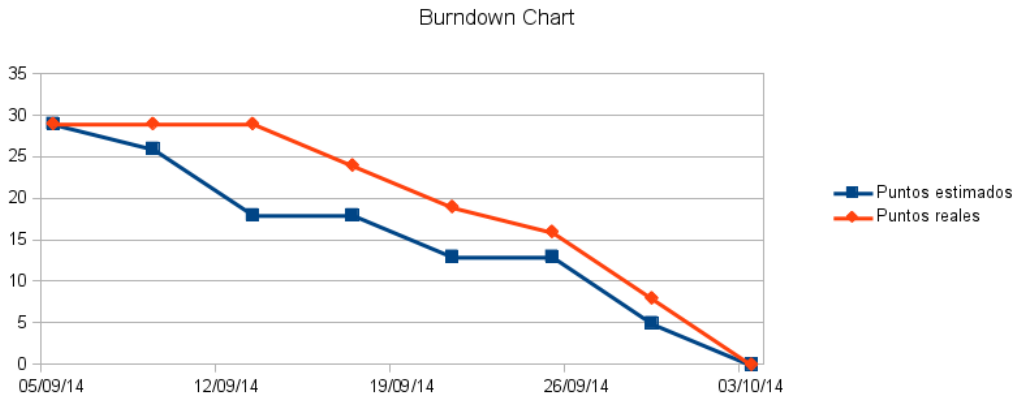






3.2.2 Avance del *sprint* en función del tiempo

Aquí se muestra como fue el progreso de las tareas a lo largo del tiempo. Podemos notar que comenzamos lento, por la cuestión de no haber hecho un buen primer *sprint*, pero al mejorarlo, logramos adaptarnos a la estimación.



4 Diseño

4.1 Diagrama de clases

El diagrama de clases puede encontrarse en el anexo de diagramas y figuras. Notar que se incluye un diagrama idéntico pero que muestra de forma aproximada como se corresponden las clases implementadas con las *stories* del *backlog*.

4.1.1 Abstracción de los sensores

El enunciado habla sobre tres tipos de sensores bien concretos. Decidimos modelar a cada uno de ellos por separado. Es decir, no realizamos una abstracción para Sensor. El motivo para sustentar esta decisión que se refleja tanto a nivel de diseño como a nivel implementativo es el siguiente.

Si bien los sensores se comportan de forma semejante, o mejor dicho, son polimórficos con respecto al mensaje `Sensar`, no lo son vistos desde la perspectiva de un lenguaje estáticamente tipado. Esto es porque cada sensor devuelve un valor de un tipo o clase distinta. Una solución a esto hubiese sido no modelar los tipos de los valores de retorno con distintas clases, y poner únicamente un valor numérico (al estilo *double*). Nos pareció que perdíamos mucha semántica con esta solución.

Otra solución podría haber sido utilizar el patrón *visitor* sobre un sensor abstracto, de manera que externamente se utilizara el sensor y se lo interpretara como la medición apropiada. Esto nos pareció que complejiza el modelo y la implementación innecesariamente, además de agregar acoplamiento entre funcionalidades claramente delimitadas.

Al no haber realizado esta abstracción se podría objetar que restringimos la extensibilidad en cuanto a más o distinto tipo de sensores. Para responder a esta posible objeción, debe observarse el modelado del `Lector Condiciones Externas`, el de los sensores y el del `Arduino`. Al desligar el sensor del `Arduino` conseguimos sensores versátiles, en el sentido de que pueden depender de uno o varios sensores reales (es decir del mundo real). Más aún, los sensores modelados permiten tener lógica interna para manejar los sensores del mundo real correctamente.

Si se quisieran agregar nuevos tipos de sensores deberían agregarse nuevas clases de sensores al modelo. Por supuesto deberá también modificarse el código de `Lector Condiciones Externas` para que se comunique con los nuevos sensores. Pero consideramos que esto es básicamente inevitable, por más que se realice una abstracción del sensor, pues la lógica de todo el sistema dependerá, en este caso, de nuevos parámetros.

4.1.2 Abstracción de los actuadores

Para el caso de los actuadores nos encontramos con una situación semejante a la recién presentada. En este caso, sin embargo, optamos por realizar una abstracción. Esta se denomina `Actuador Simple`. El nombre refleja la naturaleza sencilla de los actuadores modelados: básicamente responden al mensaje `Suministrar` con una cantidad. Donde `Cantidad` es una clase que representa valores discretos y que además son interpretados por cada actuador de forma independiente.

Esta abstracción permite realizar una calibración de cada actuador a la hora de inicializar el sistema, que queda guardada en el actuador. Por otro lado permite, al `Tomador Decisiones`, devolver decisiones en un formato semejante al almacenado en el `Plan Maestro` (y el descripto en el enunciado), que únicamente especifica cantidades aproximadas, las modeladas en la clase `Cantidades`.

4.1.3 Clase `Cantidad` y `Calibrador`

Justamente para que la cantidad tenga la semántica apropiada en cada contexto, es necesario que el actuador sepa interpretarla. Para esto se lo debe calibrar al inicializar el sistema, de esto se encarga el `Calibrador`. Además la calibración podría ser incluida en el plan maestro para soportar distintos tipos de planta de forma cómoda. Si bien esto no estaba en el enunciado y no fue implementado, el modelo resulta extensible para estas modificaciones, ya que resulta esperable que se necesite este tipo de funcionalidad en el futuro.

4.1.4 Interacción entre `Tomador Decisiones` y `Lector Condiciones Externas`

Inicialmente decidimos tener un `Timer` que periódicamente llame a `Lector Condiciones Externas` con el mensaje `sensarCondicionesExternas`. Una vez recopilada la información de los sensores, `Lector Condiciones Externas` mandaba el mensaje `tomarDecisiones` a `Tomador Decisiones`.

El problema con este protocolo es que `Tomador Decisiones` depende de `Lector Condiciones Externas` para entrar en juego. Por otro lado, `Lector Condiciones Externas` termina dependiendo de `Lector Condiciones Externas` a nivel diseño e implementación, lo cual no resulta razonable, pues son partes independientes del sistema y este acoplamiento puede ser evitado.

Para esto usamos dos `Timer`. Los objetos que antes estaban acoplados, ahora pueden actuar libremente, siendo activados por `Timer`. `Lector Condiciones Externas`, luego de sensar, escribe los resultados en el `Historial`. `Tomador Decisiones` lee estos resultados al ser activado, y toma una decisión.

Otro aspecto interesante que surgió al analizar esta interacción es el comportamiento estilo *observer* que se da entre `Timer` y `Lector Condiciones Externas` y entre `Timer` y `Tomador Decisiones`. Intentamos utilizar el patrón clásico en el diseño, pero no resultó natural. Los motivos son principalmente dos:

- El `Timer` se comporta como un observable, pero tiene una sutileza: debe ajustarse el tiempo. Si bien esto puede ser solucionado de forma prolija agregando objetos, decidimos que complicaba el diseño por una cuestión únicamente formal, que no facilitaba nada concreto.
- Siempre que se siga usando al `Timer` como tal, el diseño seguirá siendo extensible, en este aspecto. Pues la funcionalidad de `Timer` no debería cambiar, por la esencia misma de un `Timer`.

Por estos motivos, creemeos que la extensibilidad no fue restringida al no utilizar un *observer* clásico.

4.1.5 `Usuario` y `Servicios del Sistema`

Para que el sistema pueda funcionar al momento de implementarlo, nos resultó esencial desacoplar totalmente el funcionamiento automático del mismo: manejo de actuadores, recopilar información, tomar decisiones, etc. Del funcionamiento asincrónico debido al uso por parte del usuario: guardar entradas sobre la planta en el `Historial`, realizar consultas, etc.

Para esto separamos el programa en dos procesos. El cliente y el servidor. Que a su vez, dieron lugar a dos objetos: `Usuario` y `Servicios del Sistema`.

El patrón utilizado es *Façade*. El `Servicios del Sistema` es quién encapsula todo el comportamiento automático del sistema.

La comunicación con el cliente no es trivial, y se detalla su diseño a continuación.

Para ver como se implementó la comunicación efectivamente, remitirse a la sección de implementación.

4.1.6 Clase `Mensaje`

Al tener que transmitir los mensajes entre `Usuario` y `Servicios del Sistema` entre procesos distintos, resultó natural y conveniente especificar la forma y el propósito de estos mensajes. Para esto utilizamos el patrón *Command*. Esto da extensibilidad a la hora de agregar nuevas funcionalidades en `Servicios del Sistema`, pues se pueden agregar nuevos comandos/mensajes comodamente.

Como los procesos corriendo en un sistema operativo UNIX únicamente pueden comunicarse enviando *bytes*, es decir, el sistema operativo no provee niveles de abstracción para enviar objetos, la clase `Constructor de Mensaje` viene al caso. Y la abstracción del `Mensaje` resulta muy útil. El funcionamiento básico es: para comunicarse el `Usuario` con el `Servicios del Sistema` (y viceversa), el objeto crea un `Mensaje`. Luego, utilizando el método `Serializar` de `Mensaje`, consigue un *string* que representa al mensaje, y que envía mediante un *socket*.

El objeto que recibe esto reconstruye el mensaje utilizando un `Constructor de Mensaje` y luego se envía este mensaje.

Si bien el proceso puede parecer complejo, es el *tradeoff* más simple [2] para que la solución no se salga del paradigma, pero al mismo tiempo, poder separar el cliente y el servidor en procesos distintos.

Como se dijo arriba, remitirse a la sección de implementación para ver los detalles de la comunicación, incluyendo cómo se ejecuta el mensaje una vez reconstruido por el servidor, mediante *double dispatch*.

4.1.7 Clase `Historial`

El `Historial` tiene actualmente tres tipos de entradas, pero esto puede ser fácilmente modificado, gracias a utilizar herencia. La parte interesante del `Historial` es la forma en que es recorrido. Si bien esta funcionalidad no fue implementada si fue diseñada. Para recorrer el `Historial` se utiliza un *Visitor*, en este caso representado por `Recopilador` quien sabe como recorrer `Historial`.

4.1.8 Diseño del `Plan Maestro`

El `Plan Maestro` consta de distintas etapas. Cada una de estas especifica un `Nivel` para cada uno de los factores que son controlados por el sistema. El cambio de etapa depende únicamente del paso del tiempo, y no de las características actuales de la planta introducidas por el usuario.

Esta fue nuestra interpretación del enunciado, si bien, luego de la última reunión con el *product owner* se nos señaló que un mejor diseño contemplaría esta posibilidad. En la ultima sección se comenta brevemente la posibilidad de mejorar este aspecto en una próxima iteración.

4.2 Diagramas de objetos

En la sección de diagramas se pueden observar dos diagramas de objetos. Uno detalla el estado del sistema haciendo omisión de la parte que concierne al `Historial`. El segundo muestra el estado del `Historial` y algunos objetos pertinentes en un intervalo de unos pocos minutos de tiempo. Esto se debe a que el `Historial` almacena una gran cantidad de información en poco tiempo.

4.3 Diagramas de secuencia

Los diagramas de secuencia muestran las dos acciones principales que representan el ciclo de vida del sistema. Una es una iteración del `Tomador Decisiones`, la otra, una iteración de `Lector Condiciones Externas`. TODO - pintar un mapita del diagrama de clases de acuerdo a las tareas del sprint - formatear e integrar los diagramas

5 Implementación

5.1 Testing

Para la integridad del *software* Para corroborar el correcto funcionamiento del sistema se crearon objetos para representar a los sensores y que simulararn a las mediciones. De forma análoga se crearon objetos que simulen a los actuadores.

Se utilizó una *suite* de tests para verificar que las serializarciones (en el caso de la comuncicación vía *sockets*) y los mensajes y construcciones entre objetos fuesen compatibles.

Para la *Demo* Se probaron tanto instanciaciones de `Arduino` que devolvieran mediciones fijas como instanciaciones que devuelvan valores aleatorios o prefijados en archivos.

5.2 *Script* de la *Demo*

En lo siguiente se pone a prueba el ciclo de vida básico del sistema: despertar a los sensores y actuadores periódicamente, tomar decisiones y guardar todo esto en el historial.

Además se muestra cómo el usuario puede modificar el plan maestro, y, lo que es más interesante, como el sistema actua conforme a este cambio.

1. Se corre el servidor (`./server`).

[2] Que se nos ocurrió.

2. El servidor carga el plan maestro.
3. Se abre el cliente (`./client localhost`).
4. Se le pide el estado fenológico a la planta y se agregan dos anotaciones. La respuesta es que la planta se encuentra en la etapa 4, donde la humedad debe ser baja, el PH bajo y la temperatura baja.
5. El `Timer` de `Lector Condiciones Externas` se activa varias veces. Así se guardan las mediciones de los sensores y de la central meteorológica. Se observan los valores: PH bajo, Humerdad bajo, Temperatura baja y Probabilidad de lluvia 10%. Todo esto se guarda en el historial.
6. Luego de algunos sensados se activa el `Timer` de `Tomador Decisiones`[3].
7. `Tomador Decisiones` consulta el último sensado en el historial y pregunta al plan maestro la estapa actual.
8. Dados estos valores decide una cantidad para cada actuador: poco, poco, poco, nada.
9. Envía la decisión a `Manejador Actuadores`.
10. `Manejador Actuadores` se comunica con los actuadores indicando la cantidad correspondiente.
11. Cada actuador se comunica con su arduino indicando el nuevo valor de su *dimmer*.
12. El tomador de decisiones guarda la decisión tomada en el historial.
13. Se realiza una nueva ronda de sensados.
14. El usuario pide al cliente modificar el plan maestro.
15. Se modifica el estado 4 del plan. Ahora la humedad debe ser abundante, **el PH bajo** y la temperatura baja.
16. Se muestra en pantalla, del servidor, que se modificó el plan.
17. Luego de otro sensado se activa nuevamente el `Timer` de `Tomador Decisiones`.
18. Se prosigue de forma análoga pero administrando una cantidad **distinta** pues el plan es distinto.

5.3 Comunicación por *Sockets*

La comunicación entre cliente y servidor se realiza utilizando UNIX *sockets*. Cuando se crea el objeto servidor, se configura un *file descriptor* con un *socket bindeado* a la primera IP no-local que se encuentre. El servidor se pone en modo escucha, esperando una conexión TCP al puerto configurado. Al finalizar la conexión el servidor se vuelve a poner en modo escucha. Es importante notar que por más que el servidor esté esperando mensajes, simultaneamente, funcionan todos los eventos asicrónicos de timers, con lo cual, ambas funcionalidades están desacopladas.

Por la contraparte, la atención de la comunicación funciona de la siguiente manera. El servidor espera mensajes a traves del *file descriptor* mencionado antes. Cuando recibe una cadena de texto (que es lo unico que puede enviarse a traves de *sockets*) se la envía a `Constructor de Mensaje` que se encarga de parsear y reconstruir el mensaje original. Una vez reconstruido, el servidor ejecuta el mensaje, pasandose a si mismo como parámetro. Mediante este *double dispatch* el mensaje sabe que debe ejecutar cierta funcionaldidad y devolver un mensaje de respuesta. El servidor entonces envía este mensaje serializado de vuelta al cliente. El mensaje de respuesta se maneja con un mecanismo análogo.

6 Retrospectiva

6.1 Asunciones

6.1.1 Fenología

Por falta de conocimiento en el área asumimos que los estadíos de maduración de la planta estaban determinados por lapsos fijos de tiempo. Esto quiere decir que el paso de una etapa a la otra está solo controlado por el paso de los meses. Esto provoca que el ingreso de indicadores fenológicos por parte del usuario sea meramente informativo.

[3] los tiempos son $1m$ para el de `Lector Condiciones Externas` y $5m$ para el de `Tomador Decisiones`.

6.2 Inconvenientes encontrados

6.2.1 Tareas programadas en el *Sprint Backlog*

Al momento de definir las tareas no teníamos experiencia en separar y definir tareas concretas para un proyecto. A pesar de nuestro intento por modularizar las tareas y no olvidar partes esenciales resultó que básicamente obviamos las tareas de modelado y diseño. Por otro lado, las tareas de implementación resultaron modularizadas de forma poco conveniente, pues no se correspondían con las clases, que fueron diseñadas luego. Si bién el *Sprint Backlog* tuvo este inconveniente, a la hora de ponernos a trabajar notamos rápidamente el problema, y logramos distribuirnos las tareas de forma eficiente.

6.2.2 Horas estimadas de implementación

Luego de haber ajustado las tareas del *Sprint Backlog* logramos una estimación aceptable de las horas de diseño, pues habiamos experimentado cuanto nos podía tomar. Por otro lado las horas de implementación quedaron un tanto ajustadas. La diferencia total fue de 19 horas.

6.3 Próxima iteración – No implementado

6.3.1 Decisiones dependientes del *input* del usuario

En lo que se implementó las decisiones de como actuar dependen del plan maestro, y de los datos recolectados por los sensores. Las características escritas por el usuario sirven a modo de *log* para que el usuario pueda chequear la evolución de la planta.

En una nueva iteración se podría modificar el proceso de decisión para que tome en cuenta las características escritas por el usuario en el historial.

Para esto debería agregarse un nuevo tipo de entrada en el historial que permita guardar características bien definidas con un formato apropiado.

Para agregar esto al diseño bastará heredar un nuevo tipo de entrada del historial, y que `Tomador Decisiones` use esta nueva clase.

6.3.2 SMS

El diseño contempla la opción de enviar un SMS en caso de que las condiciones cambien abruptamente, esto se haría mediante un objeto encargado de leer las entradas de condiciones externas, analizarlas y obrar en caso de ser necesario. Esto fue modelado como `Encargado avisos urgentes`.

6.3.3 Implementación del `Calibrador`

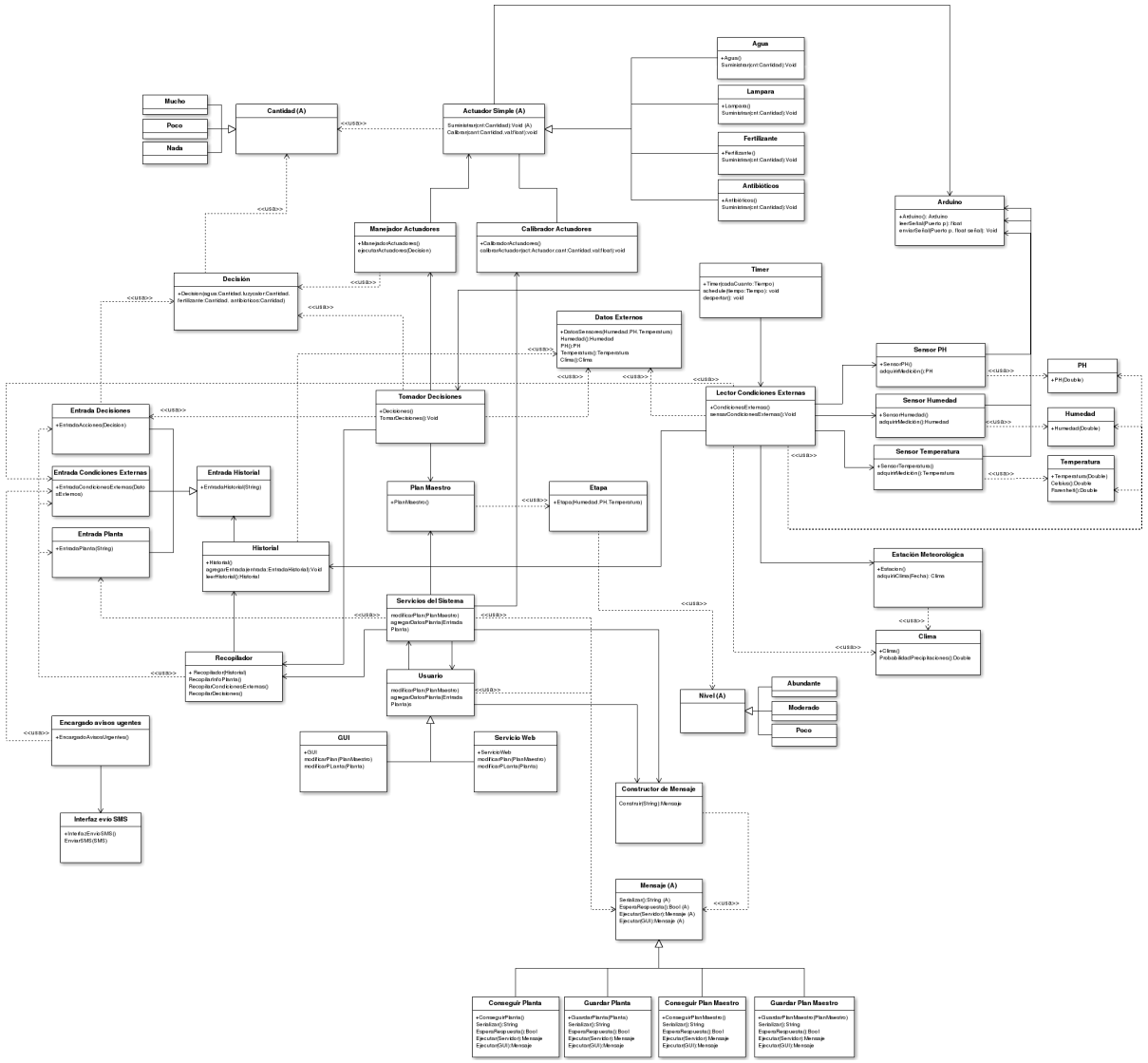
En una de las últimas reuniones con el *product owner* se nos hizo notar que no estaba contamplada la posibilidad de graduar las *Cantidades* que los actuadores deben interpretar para poder cuidar de la planta. Por ello se modificó el diseño creando un `Calibrador`, aunque el mismo no se icluyó en el *sprint*. La implementación del mismo no debería ser problemática, pues deben agregarse mensajes en `Servicios del Sistema` e implementar la clase `Calibrador`.

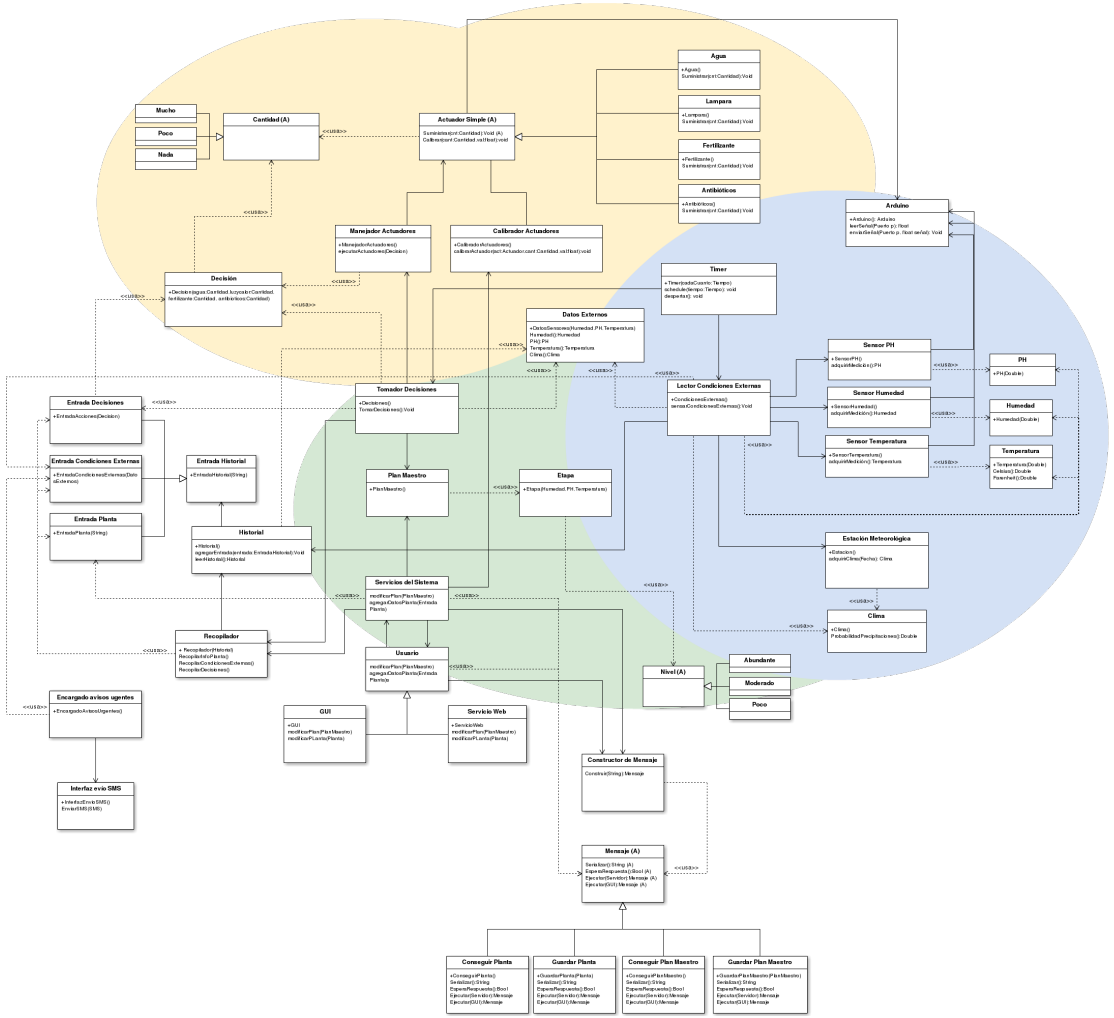
6.3.4 Implementación del `Recopilador`

Recordemos que el historial posee tres tipos de entrada distintos, es por esto que en el siguiente *sprint* se debe implementar el *Visitor* `Recopilador` El mismo se encargaría de navegar el historial y recuperar las entradas requeridas evitando romper el encapsulamiento de la implementación del `Historial`.

7 Anexo de diagramas

7.1 Diagrama de clases



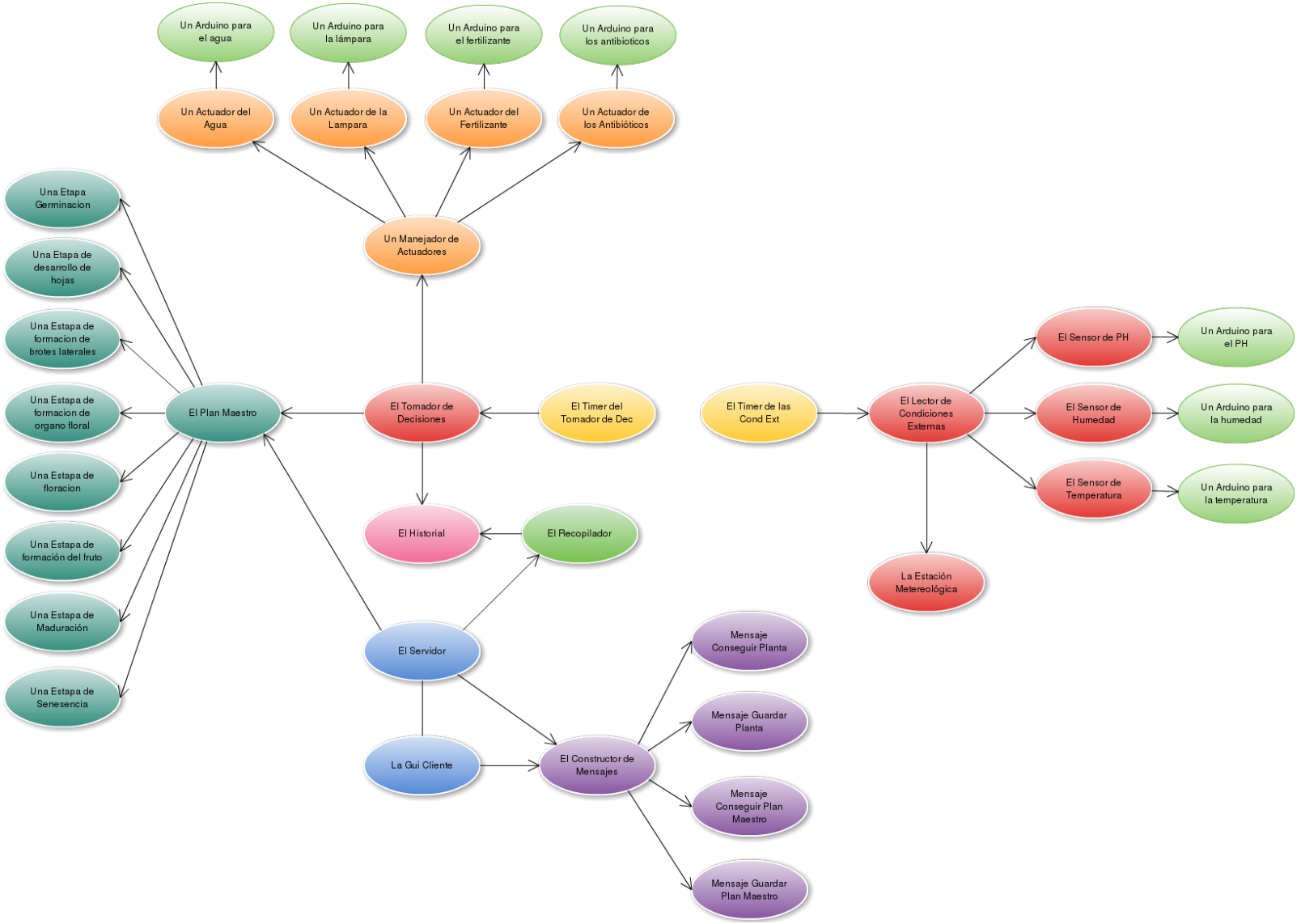


INT1 & INT2

INT12

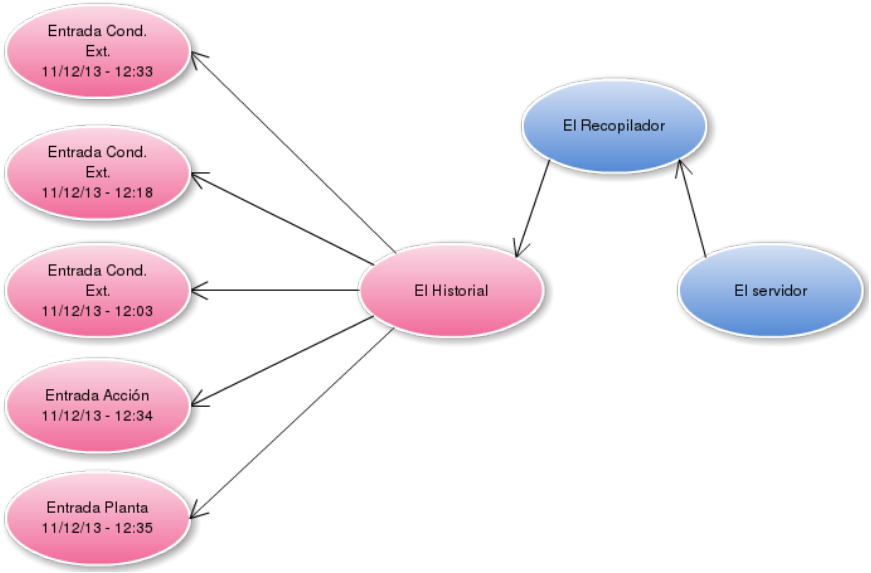
INT10 & GAL2

7.2 Diagrama de objetos sin `Historial`

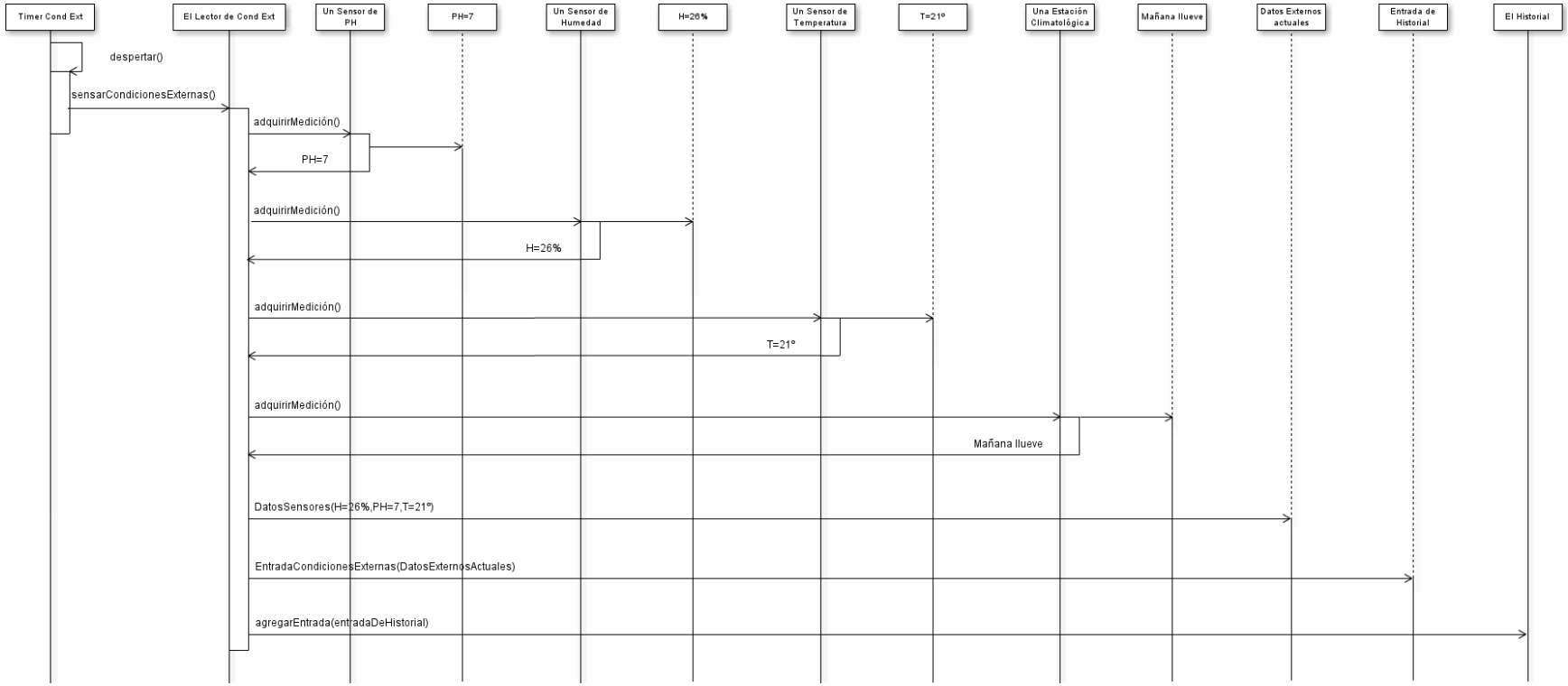


7.3 Diagrama de objetos de `Historial`

MINUTOS 12:33 a 12:35



7.4 Diagrama de secuencia de Lector Condiciones Externas



7.5 Diagrama de secuencia de Tomador Decisiones