6. ESPECIFICACIÓN

En este capítulo se detalla, de una manera más rigurosa, la definición del sistema que se va a desarrollar.

En primer lugar se comenta el diagrama de casos de uso, que sirve para delimitar las tareas del sistema bajo unos roles concretos. En este sistema el único rol del que se dispone es del rol de “Experimentador”. Se podría dar, sin embargo, que hubiera más roles, en función de las tareas asignadas.

A continuación se detalla el diagrama de dominio del sistema. En este diagrama se detalla como están estructuradas y relacionadas las diferentes partes lógicas del sistema.

Finalmente, se explican algunas de las operaciones más importantes del sistema mediante diagramas de flujo, donde se puede ver cómo se interacciona entre las partes lógicas del sistema para realizar dichas operaciones.

6.1 Diagrama de casos de uso

En el diagrama de casos de uso, se puede ver gráficamente la asignación de las tareas que debe realizar el sistema (descritas también en el análisis de requisitos) a un rol.

El diagrama se estructura de manera que se puede intuir la jerarquía y el orden de las tareas.
En el diagrama de la figura anterior se observa el rol del "Investigador". Un investigador podrá realizar todas las tareas que se observan en la ilustración.

En primer lugar podrá gestionar los proyectos y realizar varias operaciones sobre ellos (la parte roja). Alguna de las tareas de proyectos señaladas, tienen subtareas. La más importante de las subtareas es la de "Modificar los elementos de Proyecto".

La tarea "Modificar elementos Proyecto", señalada en color lila, ofrece a su vez, al investigador la posibilidad de realizar más tareas, indicada cada una de un color diferente:

- Gestionar Pasos
- Gestionar Scripts
- Gestionar Comandos
- Gestionar Variables
6.2 Diagrama de clases

Mediante el diagrama de clases se especifica la definición de las entidades lógicas que se necesitan para construir el sistema y las relaciones que hay entre ellas.

Puesto que el diagrama de clases es un poco extenso se fraccionará y ser verá por partes:

- Diagrama de clases de las clases principales
- Diagrama de clases de las clases auxiliares utilizadas
- Diagrama de clases de los controladores
- Diagrama de clases de la interfaz gráfica

<table>
<thead>
<tr>
<th>INTERFAZ GRÁFICA</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOMINIO</td>
</tr>
<tr>
<td>CONTROLADORES</td>
</tr>
<tr>
<td>DATOS</td>
</tr>
</tbody>
</table>

6.2.1 Diagrama de clases de las clases principales

Las clases principales que se van a usar son las que modelizan las entidades que intervienen a la hora de diseñar un experimento, descrito en esta documentación como “Proyecto”.

Las clases principales son:

- cProyecto: representa el experimento que se va a definir. En él se guarda la descripción, un comentario, la ruta dónde se encuentra definido todo el experimento y la relación de Pasos asignada.
- cPaso: representa, en la estructura jerárquica, la ejecución secuencial de un conjunto de scripts que tiene un objetivo común.
- cScript: es un conjunto de variables, comandos y sentencias agrupadas. Representa lo que será, posteriormente, un archivo físico.
- cComando: define una o varias sentencias. Puede utilizar variables para parametrizar ciertos parámetros de las sentencias.
- cVariable: parametriza las variables utilizadas en las sentencias.

En el diagrama mostrado a continuación se puede ver, con más detalle, todas las clases que forman el dominio de las clases principales, junto con sus atributos, operaciones y relaciones.
6.2.1.1 Atributos

Todas las clases disponen de dos atributos fijos: el identificador y la descripción.

- La clase cProyecto dispone, además de los atributos “comments” que permite escribir comentarios acerca del proyecto y “path” que es la ruta física donde se encuentra la definición de todos los objetos que definen el proyecto.
- La clase cComandoSimple dispone de un atributo “command”, que es dónde se registra toda la parte del pseudo código tcshell.
- La clase cVariable dispone de un atributo “nombre” que especifica el nombre con el que se utilizará la variable en el pseudo código tcshell y “value” que especifica que valor inicial tiene esa variable.

6.2.1.2 Procedimientos

Como procedimientos, todas las clases disponen del procedimiento “toString()” que devuelve una cadena con “id – descripción”.

- La clase cProyecto tiene el procedimiento “clone”, que a partir del identificador de un proyecto, devuelve otro que es una copia exacta del referenciado.
- La clase cPaso y cScript cuentan con un procedimiento “DeleteElementContainer” que se encarga de eliminar una de las relaciones “Pasos” y “Scripts”, respectivamente.

6.2.2 Diagrama de clases de clases auxiliares utilizadas

Con la finalidad de ofrecer unas estructuras de datos organizadas, se han definido las siguientes clases:

- cList: representa una lista genérica.
- ConfiguracionXML2Java: proporciona las funcionalidades necesarias para cargar el archivo inicial de configuración.
- clid: representa un conjunto de identificadores de objeto de una clase determinada.
- cContainer: guarda la información necesaria para mantener una relación 1 a N ó N a N de dos clases.
- cHashId: sirve para hacer el mantenimiento de los identificadores libres en el sistema en una ejecución.

Estas clases, entre sí, no guardan ninguna relación directa. Estas clases son usadas por otras clases del resto del dominio para facilitar el orden y el entendimiento estructural, como se apuntó anteriormente.

<table>
<thead>
<tr>
<th>cList</th>
</tr>
</thead>
<tbody>
<tr>
<td>elements : Vector</td>
</tr>
<tr>
<td>add(o : Object) : boolean</td>
</tr>
<tr>
<td>add(index : int, o : Object) : boolean</td>
</tr>
<tr>
<td>clear()</td>
</tr>
<tr>
<td>contains(o : Object) : boolean</td>
</tr>
<tr>
<td>get(index : int) : Object</td>
</tr>
<tr>
<td>indexOf(o : Object) : int</td>
</tr>
<tr>
<td>isEmpty() : boolean</td>
</tr>
<tr>
<td>iterator() : Iterator</td>
</tr>
<tr>
<td>remove(o : Object) : boolean</td>
</tr>
<tr>
<td>remove(index : int) : boolean</td>
</tr>
<tr>
<td>size() : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>clid</th>
</tr>
</thead>
<tbody>
<tr>
<td>tipo : String</td>
</tr>
<tr>
<td>lista : cList</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cContainer</th>
</tr>
</thead>
<tbody>
<tr>
<td>id : int</td>
</tr>
<tr>
<td>oclass : String</td>
</tr>
<tr>
<td>order : boolean</td>
</tr>
<tr>
<td>ispublic : boolean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cHashId</th>
</tr>
</thead>
<tbody>
<tr>
<td>id : int</td>
</tr>
<tr>
<td>omod : boolean</td>
</tr>
<tr>
<td>oact : boolean</td>
</tr>
<tr>
<td>oelim : boolean</td>
</tr>
</tbody>
</table>

6.2.2.1 Atributos

Los atributos de estas clases representan:
- **cList**: el atributo "elements" representa la colección de objetos de la lista.
- **cid**: está formado por el atributo "tipo", que almacena el nombre de la clase del atributo "lista". Este contiene una lista de identificadores de objetos de la clase especificada en el atributo explicado anteriormente.
- **cContainer**: sus atributos recogen la información necesaria para registrar una relación: el identificador del objeto relacionado, la clase de ese objeto, si ese objeto es visible al usuario (ya que pueden existir relaciones entre clases que no tienen porqué ser visibles).
- **cHashid**: los atributos de esta clase indican si la instancia de un identificador se ha modificado ("mod"), se ha actualizado ("act") o se ha eliminado ("elim").

### 6.2.2.2 Procedimientos

No se hará hincapié especial en este apartado, puesto que los procedimientos de la clase "cList" son los propios de una lista genérica y los procedimientos del objeto "ConfiguracionXML2Java" sólo cargan la información que se guarda en los atributos.

### 6.2.3 Diagrama de clases de persistencia

La capa de persistencia la forman dos clases:

- **DataManager**: puesto que los archivos de los que se hablan en el punto del IDManager son archivos con formato XML es conveniente que exista una clase que tenga la responsabilidad de traducir ese formato al formato de un objeto.

- **FileManager**: independientemente de lo que realice la clase DataManager y partiendo que todo el sistema está basado, a bajo nivel, en el sistema de archivos, se hace necesaria la presencia de esta clase, que realiza gestiones a nivel del sistema de ficheros.

A continuación se especifican está dos clases.

### 6.2.3.1 DataManager

La clase de este controlador tiene como función realizar de puente entre el formato XML de almacenamiento de los objetos y el objeto en sí.
El único atributo que alberga esta clase es un objeto de una librería llamada xStream de código abierto, realizada por Joe Walnes, que realiza las funciones de parseo y traducción del formato XML a objetos Java. Se puede encontrar más información de esta librería en la página web http://xstream.codehaus.org/index.html.

Esta librería ofrece muchas más posibilidades, pero para el desarrollo de este sistema sólo se han utilizado la carga y descarga simple.

Las dos operaciones de este controlador realizan las siguientes tareas:

- **read**: a partir de un path, una clase y un identificador, devuelve el objeto de esa clase con ese identificador.
- **Write**: a partir de un objeto, un path y un identificador, guarda el objeto con el identificador seleccionado en un archivo XML.

Esta clase, además, utiliza el controlador FileManager, explicado a continuación.

### 6.2.3.2 FileManager

Si toda la gestión de información se almacena en archivos, se hace necesario un controlador que gestione toda actividad con los archivos.

El FileManager tiene operaciones que son útiles para realizar toda la gestión de archivos. El siguiente diagrama muestra su estructura.

#### FileManager

- `read(pathArchivo : String) : String`
- `write(pathArchivo : String, contenido : String) : String`
- `deleteFile(path : String) : int`
- `deleteFolder(path : String) : boolean`
- `getFilesNumber(path : String) : int`
- `getFilesName(path : String) : String[]`
- `exist(path : String) : boolean`
- `CopyDirectory(orig : File, dest : File)`
- `CopyFile(orig : File, dest : File)`

Las operaciones del FileManager realizan las siguientes funciones:
• **Read**: lee un archivo a partir del path y devuelve su contenido en una cadena.
• **Write**: escribe el contenido de una cadena en el archivo indicado por el path.
• **deleteFile**: borra el archivo del path indicado.
• **deleteFolder**: borra el directorio indicado por el path.
• **getFilesNumber**: indica cuántos archivos hay en un path determinado. No tiene en cuenta el tipo de archivo, por tanto también cuenta los directorios.
• **getFilesName**: devuelve en un vector, el conjunto de nombres de archivo de un path determinado.
• **Exists**: indica si el archivo o carpeta especificada en el parámetro path, existe.
• **CopyDirectory**: copia el directorio especificado en el origen al directorio especificado en el destino.
• **CopyFile**: copia el fichero especificado en el origen al directorio especificado en el destino.

### 6.2.4 Diagrama de clases de los controladores

El sistema se organizado de manera que queda estructurado en capas. Una de las capas la forma el dominio de datos, otra la interfaz gráfica, otra la capa de persistencia capa de controladores.

Las clases que contienen los controladores ayudan a realizar tareas que son comunes para todo el resto de clases, tanto de dominio como gráficas.

Se han definido los siguientes controladores:

- **IdManager**: esta clase se encarga de realizar la gestión de identificadores de clases. Esta clase es necesaria partiendo que no se dispone de ningún sistema gestor de bases de datos y, a bajo nivel, todo se gestiona con archivos. El sistema debe ser capaz de distinguir la correspondencia entre los archivos y los objetos que se han definido.

- **CompilerManager**: la idea de separar en esta clase las funciones que generan el código de tcshell, facilita uno de los requerimientos no funcionales planteados en el análisis de requisitos: la escalabilidad.

En los siguientes subapartados se describen, con más detalle, cada uno de ellos.
La clase controladora “IdManager” tiene como función realizar la gestión de los identificadores libres.

El atributo “lista” del IdManager tiene utilidad para guardar los identificadores a bajo nivel. Sin embargo, para gestionar los identificadores en tiempo de ejecución se hace mediante una tabla hash, que viene representada por el atributo “hash”.

En la lista del atributo “lista” del objeto IdManager se guardan objetos cld. Cada uno de ellos contiene un atributo con el valor de la clase de objeto que representan los identificadores guardados en el atributo “lista” de la clase cld.

La tabla hash gestiona objetos de la clase cHashld. Estos objetos facilitan la gestión de los identificadores, asunto que se explica en capítulos posteriores.

Las operaciones utilizadas en el objeto IdManager sirven para:

- **loadHash:** carga los identificadores del atributo “lista” en el atributo “hash”.
- **reWriteList:** es la operación inversa a la anterior. Carga los identificadores contenidos en el hash a la lista.
- **getNewld:** devuelve un identificador libre de la clase especificada. Se utiliza, por ejemplo, cuando se da de alta un objeto nuevo, sin confirmar sus cambios.
- **removeld:** registra como no usado un identificador “id” de la clase especificada. Se usa, por ejemplo, cuando se elimina un objeto nuevo dado de alta con cambios o sin ellos y de objetos ya existentes.
- **updateId:** cambia el estado de los atributos de un identificador de la clase especificada. Esta operación se utiliza, por ejemplo, cuando se confirman los cambios de un objeto que había sido dado de alta.
6.2.4.2 CompilerManager

La función de esta clase controladora es de generar la estructura que contendrá el experimento especificado por la jerarquía de objetos.

La estructura será la siguiente:

- Carpeta <descripción proyecto>
  - Carpeta <paso1 descripción>
  - Carpeta <paso2 descripción>
    - Script <descripción script21>.sh
    - Script <descripción script22>.sh
    - ...
    - Script <descripción script2N>.sh
  - ...
  - Carpeta <pasoN descripción>

Esta estructura estará contenida en una carpeta inicial que indique el usuario.

Las operaciones que se realizan en esta clase son las siguientes:

- **Compile**: este es el procedimiento principal y es el que realiza la tarea de generar todo la estructura del experimento.
- **CompilePaso**: este procedimiento genera la estructura representada por un objeto Paso.
- **CompileScript**: genera el fichero que representa el script en tcshel.
- **CompileComando**: se encarga de establecer las variables y substituirlas dentro del código para que se pueda interpretar en tcshel.
6.2.4 Diagrama de clases de la interfaz gráfica

En este apartado se examina la interfaz gráfica del sistema. La forma de hacerlo será descriptiva. La forma de justificar la arquitectura del sistema se realizará en el capítulo siguiente, en el diseño.

Dividiremos en tres partes la visualización del dominio de la interfaz. Puesto que los proyectos se gestionan de forma diferente del resto de clases, una de las separaciones que se harán en el diagrama es esta.

Otra separación que se hará será la de las interfaces propias para gestionar cada clase y, por último, la separación que queda, es la de los nodos que permiten visualizar cada instancia de objeto de cada clase.

En definitiva:

- Interfaz de proyecto
- Interfaz de mantenimiento de las clases de objetos restantes
- Interfaz de nodos

6.2.4.1 Interfaz de la clase Proyecto

Como se ha visto en el diagrama de casos de uso, los proyectos son la base del resto de casos. Partiendo de este principio se ve a continuación la estructura de la interfaz.
La principal clase es "iProjects". Esta clase consta de botones que permiten realizar todas las tareas relativas a los proyectos. Puesto que es la clase principal, contiene las referencias de los principales controladores para ser usados por los objetos de dominio.

Las operaciones de esta clase realizan las siguientes tareas:

- **ButCompilarMouseClicked**: lanza una operación sobre la instancia de proyecto actualmente seleccionada e invoca el procedimiento del controlador CompilerManager.
- **jLstProjectMouseClicked**: carga una instancia del proyecto seleccionado de una lista de visualización.
- **jButDuplicarMouseClicked**: duplica la instancia de proyecto seleccionada a la ruta especificada.
- **jButCargarMouseClicked**: carga una instancia de la clase "iMain" que permite modificar todos los elementos que forman parte de un proyecto.
- **jButEliminarMouseClicked**: elimina la instancia de proyecto seleccionada.
- **jButGuardarMouseClicked**: guarda la instancia de proyecto seleccionada.
• jMenuItemPasoMousePressed: elimina la instancia de la clase Paso asignada a la instancia de la clase Proyecto.
• jMenuItemAñadirPasoMousePressed: asigna una instancia de la clase Paso a una instancia de la clase Proyecto.
• jMenuItemNuevoMousePressed: crea una nueva instancia de la clase Proyecto.
• initData: inicializa la instancia de la clase iProjects.
• LoadProjectList: carga todas las instancias de la clase Proyecto disponibles.
• LoadAvailableList: carga una lista de todas las instancias de la clase Paso disponibles.
• LoadSelectedList: carga una lista de todas las instancias de la clase Paso asociadas a la instancia de la clase Proyecto.

Como se ha ven en el diagrama anterior, la clase iProjects está relacionada con la clase iMain, que es la principal clase gráfica para gestionar las partes componentes de un proyecto.

Al ejecutar el procedimiento “jMenuItemCargarMousePressed” se crea una nueva instancia de la clase iMain. Esta clase tiene su particular importancia, y se ve con más detalle en el capítulo de diseño.

Sus operaciones son:

• initData: inicializa las instancias de las clases que forman parte de la instancia de la clase Proyecto seleccionada.
• insertTreeFile: procedimiento utilizado por initData que inserta en una estructura de árbol una instancia de una clase determinada.
• insertTreeObject: procedimiento que inserta en una estructura de árbol una instancia de una clase determinada.
• Valuechanged: procedimiento que detecta el cambio de selección de una instancia de una clase a otra instancia (de la misma clase o no).
• jMenuItemNuevoActionPerformed: procedimiento que crea una nueva instancia de una clase determinada por una selección de clase dentro de una estructura de árbol.
• jMenuItemGuardarActionPerformed: procedimiento que guarda los cambios en las instancias modificadas dentro de una estructura de árbol.
• jMenuItemEliminarActionPerformed: elimina una instancia de una clase determinada por una selección de clase dentro de una estructura de árbol.

6.2.4.2 Interfaz de nodos

La clase iMain dispone de unos procedimientos para modificar las instancias de las diferentes clases que forman un proyecto. Para poder seleccionar estas instancias, contenidas en una estructura de árbol, se define una estructura de nodos.
El diagrama que se muestra a continuación describe esta estructura, partiendo de la clase iMain:

Como se observa en el diagrama, existe una relación 1 a N entre la clase iMain y la clase cNodo (que es abstracta). Esta relación es la que posteriormente permite seleccionar las instancias de nodos de las diferentes clases heredadas.

Cada una de estas clases heredadas tendrá asociada, además, la clase con el tipo de nodo.

La clase abstracta, además facilita el procedimiento abstracto "setConcrete", que puede ser redefinido en las clases heredadas de nodo, para poder particularizar situaciones del nodo.

Como se verá en el apartado siguiente, la estructura de las clases de interfaces del resto de las entidades, sigue un mismo criterio.

6.2.4.3 Interfaz de mantenimiento de las clases de objetos restantes

Como se anunciaba ya en el anterior apartado, la estructura de las interfaces de las clases de objetos restantes (cPaso, cScript, cComando…) siguen una estructura bastante similar a la de los nodos.
El siguiente diagrama muestra la estructura de las clases de dichas interfaces:

![Diagrama de clases](image)

En este caso cada clase de interfaz (iTipoVariable, iVariable, iComandoSimple...) dispone de un procedimiento que permite cargar la información necesaria para la interfaz a partir de la clase asociada.

### 6.3 Diagramas de secuencia

Una vez establecido el diagrama de dominio se puede establecer cómo interactuan los diferentes objetos para obtener datos, calculados o de otras clases.

Para ello se dispone de los diagramas de secuencia. En esta sección se tratará uno, a modo de ejemplo, de los principales procedimientos que se realizan entre clases, ya que el resto de procedimientos son relativamente sencillos.

El procedimiento escogido a mostrar es "compile" de la clase CompilerManager.

En las páginas siguientes se especifica, mediante diagramas de secuencia, cómo es este procedimiento.
Para empezar se describe desde donde se llama a la operación “compile” del controlador CompilerManager:

Como se puede observar, este procedimiento se llama desde iProjects. Tiene dos parámetros; el path de origen (o sea, dónde están guardados los archivos que representan los objetos del modelo de datos) y el path destino (dónde se generarán los archivos con la estructura descrita anteriormente que contendrá los scripts).

Partiendo de este diagrama de secuencia inicial, se detalla cuales son los procedimientos implicados en el desarrollo de “compile”.

46
En el diagrama de secuencia siguiente se describe la operación “compile”:

Lo primero que realiza el procedimiento “compile” es obtener todas las instancias de objetos de clase “cPaso” asignadas a ese proyecto, mediante el procedimiento “pasos”.

Este procedimiento devuelve una colección de instancias y, como se ha de generar el código de cada uno de estos objetos, se necesitará un iterador.

Antes de hacer el tratamiento de cada paso, se crea una carpeta para guardar la estructura que se ha definido en el apartado del diagrama de clases del controlador de compilación (“CompileManager”). Lleva como nombre la descripción del proyecto.

A continuación se obtiene el código asociado a ese paso, o sea el de los scripts contenido por ese paso, mediante el procedimiento “compilePaso” y se reitera el proceso hasta tratar todos los pasos.

Puesto que obtener el código mediante el procedimiento “compilePaso” tampoco es trivial, se muestra a continuación su diagrama de secuencia:
Este procedimiento, por la estructura dentro del diagrama de clases, es similar al procedimiento descrito anteriormente.

Lo primero que hace el procedimiento es obtener los scripts asociados al paso. La diferencia con el anterior es que, en este punto, sí que se dispone de la instancia del objeto cPaso concreto ya que ha sido pasada como parámetro del procedimiento.

Una vez que se tiene la colección de instancias del objeto de clase cScript, o sea, todos los scripts asociados a ese paso, se hace un tratamiento parecido al que se hacía en el procedimiento “compile”.

Se genera una carpeta, con nombre la descripción de la instancia del paso que se ha pasado como parámetro del procedimiento, que servirá para guardar el código generado de los scripts vinculados a ese paso.

Una vez generada la carpeta o directorio dónde se guardarán los scripts, se va generando el código de cada script, uno a uno, mediante el procedimiento “compileScript”.

 Nótese que a todos estos procedimientos, de similares características, se les pasa como parámetros el path origen donde se encuentran almacenadas todas las instancias de objetos asociadas al proyecto o experimento y el path destino, que es la ruta donde se generará toda la estructura con el código generado.
Partiendo que el código del procedimiento “compileScript”, que aparece en el diagrama de secuencia de la figura anterior, tampoco es trivial, se detalla en el siguiente diagrama de secuencia su estructura:

La estructura de este procedimiento es similar a la del procedimiento “compilePaso”, a excepción que en el procedimiento “compileScript” ya no se genera una carpeta, puesto que se ha llegado a las “hojas” de la estructura jerárquica de directorios o carpetas.

Se obtiene, en primer lugar, todos los comandos asociados a la instancia de la clase cScript, que ha sido pasada como parámetro.

Una vez se tiene la colección de instancias de objetos de clase cComando asociada a la instancia del objeto de la clase cScript, se crea el fichero en el que se guardará el script, con una extensión “.sh”. Además, como el lenguaje del script es el tcshell, se escribirá también la cabecera.

El código que se copiará dentro de este archivo de script se tratará en el procedimiento “compileComando”, como se observa en la figura anterior.

En el diagrama que se muestra a continuación se puede ver la estructura de este procedimiento, “compileComando”, que se encarga de gestionar las variables que aparecen en el código y sustituírlas por el valor adecuado de cada variable:
Este procedimiento obtiene el atributo de la instancia de la clase cComandoSimple “command”, que contiene el código definido por el usuario desde el sistema. Las variables estarán codificadas dentro de la cadena de texto de este atributo, por tanto, se han de identificar, definir y sustituir dentro del texto antes de guardarlo en el archivo “.sh”.

El procedimiento “parsearVariables” se encarga, a partir del path origen, donde se encuentran las instancias de los diferentes objetos que forman el proyecto, de localizar, si existen, las diferentes variables utilizadas en el comando.

Una vez localizadas las guarda y las utiliza en el procedimiento “sustituirVariables”, que tiene como misión sustituir todas las variables localizadas en el procedimiento anterior por la variable real del código tcshell.

Cuando se han sustituido todas las variables existentes en la cadena del atributo “command”, se guarda el contenido de este atributo en el archivo “.sh”.

50
7. DISEÑO

Antes de la implementación del sistema, aún hace falta realizar un paso intermedio ya que la especificación no establece las tecnologías que se usarán y por tanto tampoco cómo afectarán esas tecnologías a la especificación.

En este capítulo se establecen diferentes alternativas a la construcción del sistema a partir de la especificación. Se habla de:

- La arquitectura del sistema, que define los subsistemas y componentes que forman el sistema y las relaciones entre estos. Un componente de la arquitectura es una parte del sistema encapsulada.

- Las tecnologías que se utilizan para el desarrollo de los diferentes componentes, que forman la arquitectura.

El diseño de un sistema tiene su importancia por varios motivos:

- En él se establece cual es la organización del sistema.

- Se seleccionan los elementos estructurales y sus interfaces

- Describe cual es el comportamiento de los elementos estructurales que forman el sistema

- Se decide la composición de los elementos en subsistemas a nivel superior

Para poder definir la arquitectura va a tener el sistema se establecerán que características debe tener el sistema.
7.1 Características arquitectónicas del sistema

En este apartado se definen las características arquitectónicas que debe tener el sistema, antes de decidir que patrón arquitectónico se va a usar.

Las principales características que debe reunir la arquitectura son:

- **Cambiable**: esta característica hace referencia a varias categorías:
  - **Extensible**: aumentar las funcionalidades del sistema o mejorar las ya existentes, sin tener que partir desde el inicio.
  - **Portable**: el sistema debe ser capaz de admitir cambios en las plataformas de hardware, software con la finalidad de adaptarse a las necesidades de cada momento.
  - **Mantenable**: se deben poder detectar errores y solucionarlos de manera rápida.

- **Reusable**: mediante las partes ya desarrolladas se podrán obtener nuevas o modificar existentes utilizándolas.

- **Fiable**: el sistema debe tener capacidad para la tolerancia a fallos y robustez frente a un uso incorrecto. Se debe puntualizar que, como los datos no están encriptados, si no en texto plano, y el usuario tiene la posibilidad de modificar la información de manera directa, el uso incorrecto se definirá en todos los cambios realizados desde el sistema.

- **Probable**: el sistema debe ofrecer la posibilidad de crear juegos de pruebas sencillos.

- **Usabilidad**: la facilidad de uso del sistema y la facilidad de aprendizaje del funcionamiento deben ser una de las características deseadas del sistema.

7.2 Arquitectura del sistema

Teniendo en cuenta la especificación y su estructuración se ha convenido que la mejor arquitectura para el sistema es una arquitectura de componentes en capas.

Esta decisión viene condicionada por los siguientes motivos:

- Se diferencian claramente diferentes capas dentro del sistema. De hecho, el capítulo anterior se intentó explicar teniendo en cuenta esta organización.

- En cada capa están definidas las funcionalidades a realizar, con lo cual es lógico que se definen estas funcionalidades como componentes independientes.
• Al definir los componentes como independientes, permiten ser reutilizados, si fuera necesario.

• La unión de todas las capas, cada una con sus componentes, forman un todo.

Atendiendo a que hay muchas definiciones de lo que es un componente, se da una definición, para evitar ambigüedades. Esta definición está basada en [Brown, 1999]:

Un componente software es un fragmento de un sistema software que puede ser ensamblado con otros fragmentos para formar piezas más grandes o aplicaciones completas.

La definición se basa en tres perspectivas del componente:

• Perspectiva de empaquetamiento: considera un componente como una unidad de empaquetamiento, distribución o entrega.

• Perspectiva de servicio: considera un componente como un proveedor de servicios.

• Perspectiva de integridad: considera un componente como un elemento encapsulado.

Otro punto a tener en cuenta es, que como requisito, no se ha contemplado que el sistema fuese distribuido (porque no ha sido requerido). Es por este motivo que la arquitectura no tiene porque estar enfocada hacia estas características.

De todas maneras, se ha de comentar que no sería muy costoso adaptar la aplicación para que fuese distribuida ya que la única capa que es susceptible de poder ser distribuida es la de datos.

7.3 Tecnologías para el desarrollo

Mediante este apartado se justificará la elección de las tecnologías escogidas para desarrollar el sistema.

Básicamente, y por las especificaciones expuestas, se deberá definir los siguientes dos puntos:

• Gestor de base de datos a utilizar

• Lenguaje de la aplicación
7.3.1 Entorno de ejecución

El entorno de ejecución, a priori debe contemplar las siguientes características:

- El sistema se va a usar en un entorno Unix.
- Es deseable que la información del sistema pueda ser consultada y modificada mediante un simple editor de textos.
- Se ha de prever que el sistema se pueda usar en otro entorno de hardware (por ejemplo, en un sistema Windows).
- Versatilidad en la localidad de la información.
- No se necesiten grandes requisitos para ejecutar el sistema.

Teniendo en cuenta los puntos anteriores, se deberá escoger unas tecnologías que los permita.

7.3.2 Gestor de datos para el sistema

El gestor de datos es un punto importante a tener en cuenta en el desarrollo de un sistema, puesto que establece la tecnología que gestionará los datos.

En este caso se podrían haber usado gestores de bases de datos adaptados para el entorno Unix (como indica uno de los puntos del entorno de ejecución). Entre ellos se podría haber escogido entre:

- Postgres
- Mysql
- Oracle
- Informix
- Etc.

La ventaja de estos gestores de datos son muchas, puesto que ofrecen operaciones para gestionar los datos probados y, normalmente, gestionan también los recursos hardware de una manera eficiente.

Sin embargo, si se utiliza uno de estos SGBD’s para el sistema a desarrollar, se tiene que algunos de los puntos del entorno de ejecución no se cumplen.

Concretamente no se cumpliría el poder consultar y modificar los datos con una simple herramienta de texto, puesto que la mayoría de estos sistemas gestores de base de datos, encriptan de una manera determinada los datos que contienen, aunque físicamente, dentro del sistema, estén presentes en simples archivos.
Otro punto que no se cumpliría es que no se necesiten grandes requisitos para la ejecución del sistema que se está desarrollando, puesto que, partiendo que el sistema no es distribuido, el hecho de ejecutar el sistema con alguno de esos SGBD’s implicaría tenerlo instalado en alguna máquina; bien una instalación local, bien una instalación por red, que hace el sistema, entonces, dependiente de algún servidor que contenga el SGBD escogido.

Vistos los inconvenientes para los SGBD’s expuestos, se plantea la necesidad de encontrar otra manera de guardar los datos en el sistema.

Inspeccionando las posibilidades, se ofrecen varias:

- Guardar la información en archivos de texto plano
- Guardar la información en archivos de texto plano organizando los datos con alguna codificación propia
- Guardar la información en archivos de texto plano organizando los datos con alguna codificación existente.

Guardar la información en archivos de texto plano sin ninguna codificación no es viable, puesto que no permite estructurar los datos que se guardan y complicaría la operación de recuperar y guardar los datos.

Partimos de dos alternativas pues. La segunda alternativa implicaría desarrollar un sistema de codificación propio para guardar los datos. Esta alternativa es correcta, pero ofrece un inconveniente. Puesto que el sistema de codificación de datos es propio, se deberá hacer un proceso de desarrollo paralelo, con sus fases de análisis de requisitos, especificación, diseño e implementación para poder gestionar los datos.

La tercera alternativa ofrece lo mismo que la segunda, pero con una diferencia: se ahorra el tiempo del desarrollo paralelo porque ya existen componentes, librerías... que realizan las operaciones de gestionar los datos.

7.3.2.1 XML

Finalmente se decide que para guardar los datos se utilizará el lenguaje XML (eXtended Markup Language).

XML es un metalenguaje desarrollado por W3C (World Wide Web Consortium), simplificación y adaptación del lenguaje SGML (Standard Generalized Markup Language), utilizado para la organización etiquetado de documentos.

El XML ofrece unas ventajas respecto a otros sistemas que son:
• XML permite transferir datos de una aplicación a otra de manera que, si ambas leen el formato, se puede reconstruir la misma información en ambas aplicaciones.

• Mover los datos en XML es sencillo, pues los datos y la codificación utilizada para guardarlos se ofrece en texto plano.

• Permite, además, definir analizadores simples o utilizar analizadores ya implementados.

• Se puede validar los errores de sintaxis, al estar basado en una gramática.

• Aunque con cualquier editor de texto se pueda generar un archivo XML (bien formado), existen herramientas que permiten hacerlo fácilmente.

• Es un metalenguaje gratuito (por contraposición de algunos SGBD's).

Vistas las ventajas, también se ha de hablar de algunos de sus inconvenientes:

• Se deberá implementar un componente que gestione la escritura y lectura de los datos en el formato XML.

• Puesto que no se utiliza un SGBD se deberá tener en cuenta la gestión de identificadores, para poder identificar los objetos guardados con el formato XML.

Otros puntos relacionados con la gestión de este metalenguaje, particularizados para este sistema son:

• Se establece que se tendrá un archivo XML por cada instancia del objeto del sistema.

• Además se establece también que el nombre de un archivo XML se formará mediante "<nombre de la clase>_<identificador>.xml".

• Para diferenciar las instancias de los objetos de cada clase y mantenerlos de una forma más estructurada, existirá una carpeta o directorio con el nombre de la clase que contendrá las instancias de objetos de esa clase.

Establecido cual es el sistema para gestionar los datos se decide en el siguiente apartado qué lenguaje de implementación se utilizará para desarrollar el sistema. En esta elección se deberá tener en cuenta la compatibilidad del lenguaje con XML.
7.3.3 Lenguaje de implantación para el sistema

Tras decidir con qué se va a gestionar la información que recoge el sistema, en este apartado se decide con qué lenguaje de programación se va a realizar el desarrollo del sistema.

Como en el caso de la gestión de datos, se pueden encontrar varias alternativas. Las alternativas, sin embargo, se han de adecuar a las características del entorno de ejecución.

Se han seleccionado las siguientes opciones:

- Tcl / Tk
- C++
- Perl
- Java

Ante las opciones anteriores y después de mucha reflexión, se ha optado por escoger el lenguaje Java.

Este lenguaje ofrece las siguientes características:

- Es multiplataforma.
- Es orientado a objetos.
- Permite realizar interfaces gráficas relativamente sofisticadas en un tiempo relativamente pequeño.
- Es adecuado para la envergadura del sistema que se va a construir.
- Es gratuito.

El lenguaje Java es uno de los más utilizados hoy en día. Prácticamente todas las argumentaciones que se den para justificar la elección de este lenguaje pueden ser argumentadas por el resto de seleccionados en la lista anterior.

Pero Java tiene una ventaja respecto a los otros: mediante un cambio simple se puede integrar en la web, simplemente haciendo del componente principal un applet.

Además de todas las razones expuestas anteriormente, también condiciona un interés personal por explorar la potencia de este lenguaje de programación en este proyecto.