EXTRACTO

Parte del desarrollo orientado a objetos es la generación de clases que mapeen de las entidades existentes en la base de datos. Esta es una práctica ampliamente difundida en varios lenguajes, como JAVA y PHP. Aunque el trabajo de realizar estas clases a mano se vuelve tedioso y agotador.

Sin embargo, en ciertos lenguajes como en JAVA, existen herramientas para la generación automática de este tipo de clases. La creación automática de las clases que se mapean de la base de datos, ayuda a prevenir errores de sintaxis por fallo humano, así como los errores de nombrado y además mejora la estandarización del código, puesto que al generarse automáticamente se obliga a que el código cumpla con los estándares requeridos para el caso.

El mapeo de entidades de la base de datos hacia clases es la base fundamental para la aplicación posterior de herramientas de programación orientada a objetos como el mapeo relacional de objetos, ORM por sus siglas en inglés. En este modelo, a partir de clases entidad se llega a establecer la relación entre los objetos mapeados y se puede implementar de manera natural el obtención de información tal como se hace dentro de las bases de datos relacionales a través de lenguaje de manipulación de datos como SQL.

De esta forma se delega el manejo de integridad de datos (de entidad y referencial)1 al ORM y el desarrollo se vuelve más limpio y enfocado a implementar reglas de negocio y no a soportar y validar consistencia de datos.

Por lo tanto, es comprensible que se necesiten herramientas similares en otros lenguajes de programación, como lo es PHP, que actualmente poseen limitadas herramientas para la generación de clases que reflejan las entidades de la base de datos que serán usadas en la aplicación.

Hay que considerar, que los artefactos que se generen deben cumplir con las mejores prácticas del desarrollo de software. Se debe conseguir un diseño robusto y extensible, capaz de soportar altas cargas de trabajo con buen desempeño y con la flexibilidad

[1] En bases de datos relacionales la integridad de entidad se refiere a que cada registro que se guarda dentro de la BDD debe ser identificado de manera única para evitar redundancia, por otro lado la integridad referencial protege que la información no afecte a los dependientes si es que se elimina o se actualiza.

suficiente para que la librería pueda adaptarse a las necesidades del proyecto en el que sea utilizada.

La forma más conveniente de conseguir estas características es mediante la utilización de patrones de diseño y metodologías de desarrollo probadas y eficaces. Para este caso en particular resulta casi instintivo el uso de patrones constructores como “Abstract Factory”.

PONTIFICIA UNIVERSIDAD CATÓLICA DEL ECUADOR

FACULTAD DE INGENIERÍA

ESCUELA DE SISTEMAS

CONSTRUCCIÓN DE UNA LIBRERÍA PARA GENERACIÓN AUTOMÁTICA DE CLASES PHP BASADA EN PATRONES DE DISEÑO UTILIZANDO PSP

TORRES MORALES JOSÉ LUIS

TRABAJO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO DE SISTEMAS

DIRECTOR: ING. FABIÁN DE LA CRUZ

QUITO, 2014

```
$.get('professor/knowledge/', function (data) {
   data = JSON.parse(data);
   $.each(data.info, function (chunk) {
      $('#this .container').append(chunk);
   });
});
$('.family').each(function (idx, member) { thank(member); });
$(this).on('end', function () { startWalking(); });
```

¡Gracias a todos!
El camino apenas empieza…

Índice de Contenidos

Índice de Ilustraciones

Índice de Tablas

CAPÍTULO 1 INTRODUCCIÓN

En este capítulo se desarrollan las razones y motivaciones de realizar este trabajo, así como el alcance del mismo.

1.1 Justificación

Parte del desarrollo orientado a objetos es la generación de clases que mapeen de las entidades existentes en la base de datos. Esta es una práctica ampliamente difundida en varios lenguajes, como JAVA y PHP. Aunque el trabajo de realizar estas clases a mano se vuelve tedioso y agotador.

Sin embargo, en ciertos lenguajes como en JAVA, existen herramientas para la generación automática de este tipo de clases. La creación automática de las clases que se mapean de la base de datos, ayuda a prevenir errores de sintaxis por fallo humano, así como los errores de nombrado y además mejora la estandarización del código, puesto que al generarse automáticamente se obliga a que el código cumpla con los estándares requeridos para el caso.

El mapeo de entidades de la base de datos hacia clases es la base fundamental para la aplicación posterior de herramientas de programación orientada a objetos como el mapeo relacional de objetos, ORM por sus siglas en inglés. En este modelo, a partir de clases entidad se llega a establecer la relación entre los objetos mapeados y se puede implementar de manera natural el obtención de información tal como se hace dentro de las bases de datos relacionales a través de lenguaje de manipulación de datos como SQL.

De esta forma se delega el manejo de integridad de datos (de entidad y referencial)[1] al ORM y el desarrollo se vuelve más limpio y enfocado a implementar reglas de negocio y no a soportar y validar consistencia de datos.

Por lo tanto, es comprensible que se necesiten herramientas similares en otros lenguajes de programación, como lo es PHP, que actualmente poseen limitadas

[1] En bases de datos relacionales la integridad de entidad se refiere a que cada registro que se guarda dentro de la BDD debe ser identificado de manera única para evitar redundancia, por otro lado la integridad referencial protege que la información no afecte a los dependientes si es que se elimina o se actualiza.

herramientas para la generación de clases que reflejan las entidades de la base de datos que serán usadas en la aplicación.

Hay que considerar, que los artefactos que se generen deben cumplir con las mejores prácticas del desarrollo de software. Se debe conseguir un diseño robusto y extensible, capaz de soportar altas cargas de trabajo con buen desempeño y con la flexibilidad suficiente para que la librería pueda adaptarse a las necesidades del proyecto en el que sea utilizada.

La forma más conveniente de conseguir estas características es mediante la utilización de patrones de diseño y metodologías de desarrollo probadas y eficaces. Para este caso en particular resulta casi instintivo el uso de patrones constructores como “Abstract Factory”

1.2 Alcance

El presente trabajo finalizará con la entrega de una librería funcional que permita la generación automatizada de clases en PHP para el gestor de base de datos MySQL.

CAPÍTULO 2 FUNDAMENTO TEÓRICO

En este capítulo se desarrollan los conceptos básicos que se deben tener en mente al momentos de desarrollar un producto, ideas que sirven de base para la creación de un sistema, librería o cualquier producto de software cuyo objetivo es ser funcional y robusto.

2.1 PHP (PHP: Hypertext Preprocessor)

Es un lenguaje open source[2] de scripting[3] de propósito general ampliamente utilizado y diseñado especialmente para desarrollo web, que puede ser embebido dentro de HTML conjuntamente con JavaScript y otras tecnologías. Su sintaxis nace a partir de lenguajes clásicos como C, Java y Perl, lo que lo hace fácil de aprender y para muchos inclusive familiar a primera vista.

El objetivo principal de los diseñadores del lenguaje es que permita a los desarrolladores escribir páginas web generadas dinámicamente de manera rápida, sin truncar el potencial del lenguaje, que permite generar muchos otros tipos de aplicaciones.

PHP surge inicialmente como un conjunto de binarios CGI[4] escritos en C por Rasmus Lerdorf , para el seguimiento de visitantes a su curriculum vitae online y los apodó “Personal Home Page Tools”. Con el tiempo los requerimientos funcionales fueron creciendo y pronto *PHP Tools* fue reescrito para brindar una implementación mucho mayor y más rica.

Esta nueva implementación, tal como se ve en la ilustración 1, era capaz de interactuar con bases de datos, proveyendo de esta manera, una herramienta mediante la cual los desarrolladores podrían implementar sencillas páginas web dinámicas, como blogs o libros de visitas. Es así que en 1995 Rasmus libera el código de PHP Tools al público[5]

[2] Software que puede ser usado, cambiado y compartido de manera libre. (http://opensource.org).

[3] Un lenguaje de scripting no necesita ser compilado para ser ejecutado.

[4] Por su nombre en inglés “Common Gateway Interface”. Estándar de comunicación entre un cliente web y un programa ejecutado en un servidor.

[5] Anuncio de la liberación de PHP Tools: *https://groups.google.com/forum/#!msg/comp.infosystems.www.authoring.cgi/PyJ25gZ6z7A/M9FkTUVDfcwJ*

para que sea utilizado a conveniencia y además para que la comunidad sea capaz de revisarlo y corregir bugs en caso de ser necesario.

```
<!--include /text/header.html-->

<!--getenv HTTP_USER_AGENT-->
<!--ifsubstr $exec_result Mozilla-->
  Hey, you are using Netscape!<p>
<!--endif-->

<!--sql database select * from table where user='$username'-->
<!--ifless $numentries 1-->
  Sorry, that record does not exist<p>
<!--endif exit-->
  Welcome <!--$user-->!<p>
  You have <!--$index:0--> credits left in your account.<p>

<!--include /text/footer.html-->
```

Ilustración 1 Sintaxis original PHP Tools (Mehdi Achour 2013)

Aunque la comunidad no vio en PHP Tools una opción fuerte como un lenguaje de programación, si tuvo mucha acogida como CGI, sin embargo, esto no duró mucho. En octubre de 1995 Rasmus lanzó una completa reedición del lenguaje aplicando deliberadamente una estructura similar a la de C y Perl para que fuera muy sencillo adoptar este lenguaje para los desarrolladores.

Rápidamente PHP Tools fue evolucionando de una librería o suite de herramientas a un lenguaje de programación completo. Se agregó soporte para múltiples bases de datos, uso de cookies, implementación de funciones definidas por el usuario, entre muchas otras.

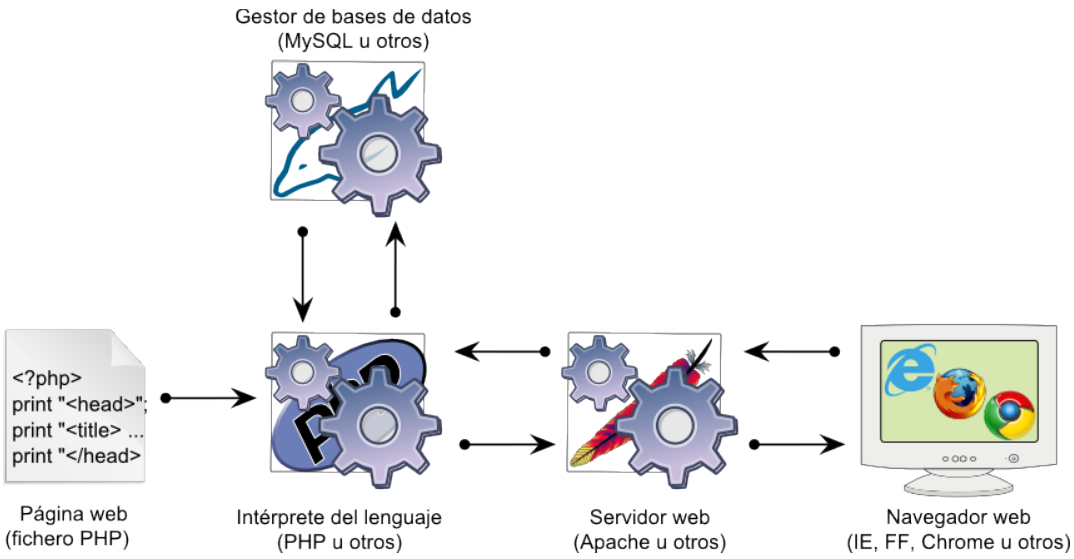

Ilustración 2 Esquema básico de funcionamiento de PHP (Marco 2013)

Para invierno de 1998 PHP estaba en su versión 4 y se consagraba como un lenguaje de programación completo y robusto para desarrollo web, seguido unos años después por la versión 5 que implementaba un nuevo modelo orientado a objetos. Con esta última versión y la incorporación del Zend Engine[6] PHP se convirtió en el lenguaje más usado alrededor del mundo con cientos de millones de servidores instalados en toda la web.

En la ilustración 2 se muestra la arquitectura básica que tiene una aplicación escrita en PHP. Hay que mencionar, sin embargo, que esta estructura generalmente se complementa con otras herramientas, tanto del lado del servidor como del cliente para construir un servidor potente que pueda soportar las cargas de transporte de datos que se requieren en la actualidad.

PHP es un lenguaje intrínsecamente flexible, no restringe al desarrollador en declaración de variables, en tipos de datos, en declaración de funciones ni en paradigmas de desarrollo, además que es un lenguaje abierto a la comunidad y con una variedad muy extensa de funcionalidades agregadas y parchadas.

Es por esto que no existen normamientos claros sobre el lenguaje, las convenciones de desarrollo en PHP son pocas o casi nulas, a diferencia de otros lenguajes como JAVA que poseen amplia documentación sobre las convenciones de nombrado y utilización del lenguaje.

A manera de tradición los desarrolladores que utilizan PHP se han basado en las convenciones utilizadas por grandes proyectos de empresas reconocidas a nivel mundial, muchas veces se utilizan convenciones internas de cada empresa o grupo de desarrollo y también hay casos en los que se utilizan adaptaciones de convenciones de otros lenguajes.

Los estándares de codificación para el presente trabajo serán presentados a detalle en el capitulo 3, en el que se detalla la metodología a seguir.

[6] Zend Engine es el motor de scripting open source que interpreta el lenguaje PHP.

2.2 DAO (Data Access Object)

Objeto de acceso de datos o DAO por sus siglas en inglés es un patrón de diseño ampliamente usado cuando una aplicación debe persistir información, sea en una base de datos, en archivos planos o en cualquier otra forma de almacenamiento de información.

El objetivo de este diseño es separar el dominio del problema del mecanismo de persistencia de datos. Es decir, nos permite desarrollar la lógica del negocio independientemente de cuál estructura de almacenamiento de datos esté siendo utilizada[7].

La aplicación de este patrón de diseño es simple: se crean interfaces que definan los métodos de manipulación a los datos y se reemplaza la implementación de acuerdo a la necesidad del producto. Mediante la inyección de implementación se puede utilizar diferentes métodos de persistencia de datos como objetos en memoria, archivos planos, archivos binarios, conexiones directas a bases de datos o APIs de persistencia.

En la ilustración 3 se expone el diagrama de clases para el patrón DAO de una entidad "Libro", suponiendo dos implementaciones concretas para persistencia en base de datos y en archivos planos.

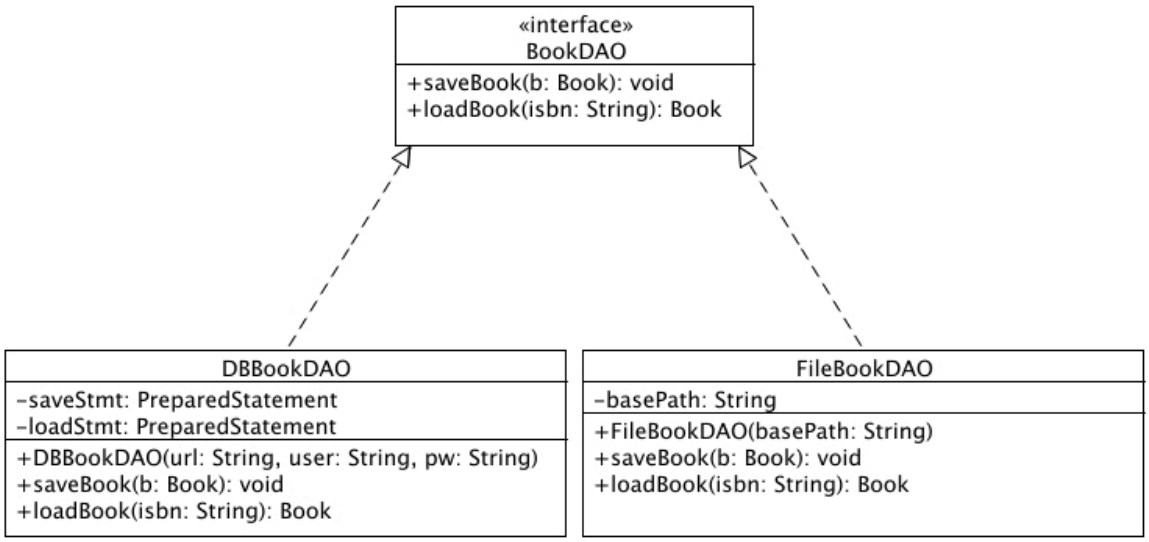

Ilustración 3 Ejemplo de Diagrama UML del patrón DAO (ISEG 2013)

[7] (Williams Michael 2012)

Es recomendado utilizar un patrón DAO cuando[8]:

- Se necesita almacenar o leer datos de una fuente repetidas veces.
- Existe la posibilidad de que la fuente de datos cambie en el futuro.
- Se requiere separar la fuente de datos de la lógica del negocio.
- Se trabaja en equipo y se necesita estandarizar el acceso a los datos.

Este patrón suele combinarse con el de la Fábrica Abstracta[9] para brindar un mayor encapsulamiento de la implementación. Así se logra tener un solo punto de la aplicación que depende de una implementación concreta y el resto de la aplicación solamente depende de los métodos públicos de un objeto constructor.

Los beneficios de usar el patrón de diseño DAO son claros: la mantenibilidad de la capa de datos se incrementa notablemente. Asimismo, el producto mantendrá un acoplamiento más bajo y la cohesión del código será mayor.

Uno de los objetivos de la programación orientada a objetos es la creación de objetos con una única responsabilidad. Separando las implementaciones de acceso a datos, los objetos del dominio del problema serán más limpios y se enfocarán solamente en la lógica del negocio y no en los mecanismos de persistencia de la información.

Entre los requisitos para implementar este patrón cumpliendo los principios de la Programación Orientada a Objetos (POO) es tener las tablas de la base de datos (BDD) mapeadas dentro de la aplicación como clases entidad. En ese punto es donde la librería que será generada pasa a jugar un papel protagónico.

Entonces, la generación automática de clases prestará una base para el desarrollo extensible de una capa abstracta de manipulación de datos. Esta capa depende de la arquitectura y del diseño del sistema, pero, de manera intuitiva y prestando atención a las mejores prácticas de la ingeniería de software la elección más lógica es el patrón DAO.

[8] (ISEG 2013)

[9] Sobre el patrón de la Fábrica Abstracta se detallará más adelante en el capítulo en el punto 2.5.1.

Es decir, la librería producto de este trabajo será la encargada de generar todo el mapeo necesario dentro de la aplicación para, a partir de ello, continuar con la creación de una aplicación, cualquiera que sea, basada en las mejores prácticas de desarrollo. Asegurando así el cumplimiento de altos estándares de calidad como lo son la mantenibilidad, desacoplamiento, alta cohesión, visibilidad y trazabilidad.

2.3 MySQL

La gran cantidad de información que se maneja en estos días empujó el desarrollo de la tecnología a buscar formas más eficientes y seguras de almacenar todos estos datos, es así que se desarrollan las bases de datos. Sin embargo, existen muchos motores de bases de datos y cada uno con diferentes perspectivas, manejos de roles y seguridades, con diferentes capacidades y limitaciones.

Para poder responder por qué se ha escogido MySQL como Sistema Gestor de base de datos, DBMS por sus siglas en inglés, hay que aclarar varios conceptos fundamentales. Términos como base de datos, tipo de BDD, DBMS que aunque de uso común, no siempre están del todo claros.

Base de datos en un término difuso que puede referirse a una lista de direcciones almacenadas en una hoja de cálculo de Excel o llegar a niveles muy altos de especialización y representar a un sistema gestor de base de datos como lo son MySQL, Oracle o cualquier otro de ellos comercial o no comercial.

En el uso común, el término BDD se utiliza para denominar al conjunto de datos en sí mismos, a los archivos resultantes de almacenar la información en un medio, al DBMS o inclusive al cliente que accede a la información. Pero en definitiva, una base de datos es una colección de datos que usualmente están almacenados en uno o más archivos[10].

Dentro de los tipos de bases de datos la que surge a la vista como la más exitosa es la de tipo relacional. Este tipo se caracteriza por organizar la información en tablas, entidades que agrupan datos que poseen características similares, y generar relaciones entre ellas, es decir, se entrelazan las tablas a través de sus atributos.

[10] (Michael 2005)

Esta es una tarea compleja y abstracta por la que un desarrollador no debe preocuparse, para ello existen sistemas gestores de bases de datos (DBMS). Las tareas de un DBMS incluyen, no solamente almacenar los datos, sino también brindar seguridad, procesamientos de queries y análisis de la información que se guarda.

A pesar de que las bases relacionales son las más conocidas y ampliamente utilizadas existen otros tipos que ofrecen características diferentes. Un tipo importante que merece ser mencionado son las bases de datos orientadas a objetos[11].

Estas permiten almacenar objetos directamente dentro de ella. Por lo tanto, no existe la necesidad de realizar procesos de conversión desde objetos de una aplicación a un formato adecuado para la base de datos.

Sin embargo, el tipo relacional es especialmente adecuado para modelar y diseñar procesos de negocio y será el tipo utilizado para este trabajo. Además, el proceso de conversión entre objetos de la aplicación y un formato para almacenar los datos, le corresponde al API de persistencia y no será discutido en profundidad porque no está dentro del alcance de este trabajo.

MySQL es una BDD de tipo relacional muy conocida a nivel mundial y que es especialmente discutida en ambientes académicos. Se han levantado con el tiempo grandes pasiones alrededor de este DBMS al punto de tener puntos de vistas totalmente opuestos dependiendo de si se está a favor o en contra de ella.

Por un lado se dice que es la base de datos más rápida, más confiable, en definitiva mejor que cualquier otra en el mercado[12] y por otro, los oponentes, la cuestionan al punto de sostener que no es ni siquiera una BDD relacional. Los hechos muestran que el número de usuarios de MySQL incrementa cada instante por todo el mundo y que la increíble mayoría están más que satisfechos.

Además, un buen punto de referencia es qué clientes (empresas) utilizan esta base de datos y qué tan eficaces y eficientes son brindando sus servicios, si a ellos les va bien

[11] Existen otros tipos de BDD que no serán mencionados debido a la naturaleza exógena en relación al objetivo del trabajo actual.

[12] (Michael 2005)

entonces MySQL no es una mala alternativa. Gigantes comerciales como Adobe, VMware, PayPal, Wikipedia, Twitter y otros la utilizan y por lo tanto se puede decir que MySQL es lo suficientemente buena para el caso[13].

Los benchmarks realizados comparando MySQL en sus versiones 5.5 y 5.6 con respecto a la velocidad de lectura y escritura de datos arrojan resultados sorprendentes, como se muestra en las ilustraciones siguientes[14]:

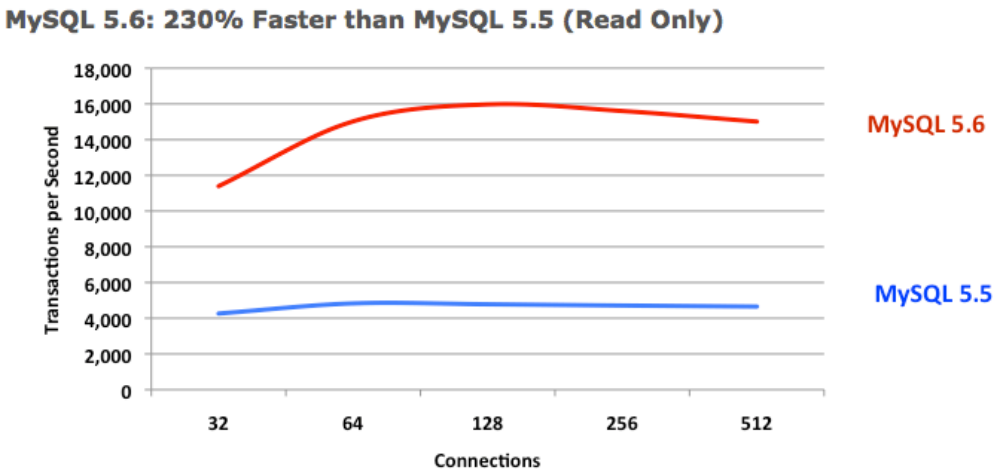

Ilustración 4 Transacciones por segundo por número de conexiones en modo solo lectura

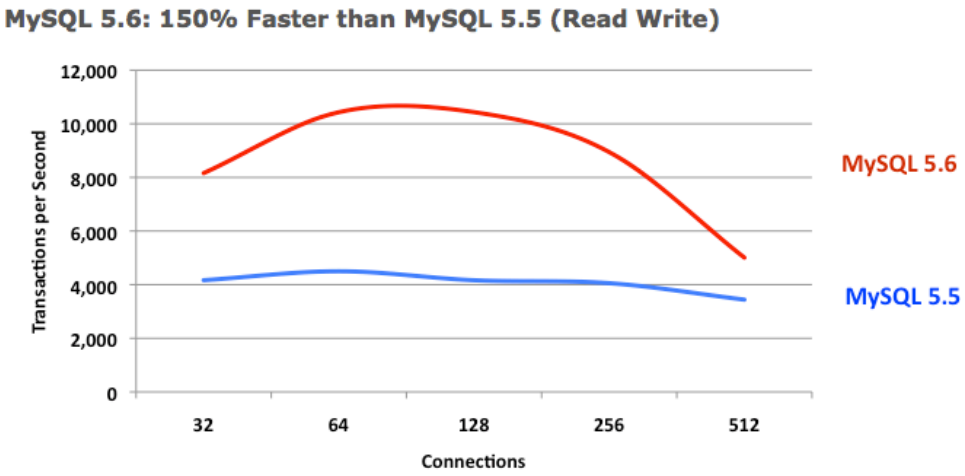

Ilustración 5 Transacciones por segundo por número de conexiones en modo lectura escritura

Por supuesto, la falta de ciertas características dentro de MySQL, que en otros DBMSs comerciales no se pasan por alto, es real. Por este motivo se analizarán de forma breve algunos de los beneficios y limitaciones del gestor de bases de datos escogido.

[13] Para una lista detallada de clientes de MySQL: http://www.mysql.com/customers/

[14] Los resultados fueron tomados de: https://www.mysql.com/why-mysql/benchmarks/. Accedido el 12 de abril de 2014.

Entre las capacidades de MySQL es la arquitectura cliente/ servidor, lo que significa que un determinado número de clientes puedan comunicarse con el servidor para obtener datos de manera recurrente. Además, es compatible con el lenguaje SQL propio de bases de datos necesario para realizar operaciones de manipulación de información dentro del DBMS: inserción, actualización, eliminación y búsqueda.

El estándar actual de SQL, es la versión SQL:2008, la cual está soportada según el manual oficial "MySQL Standards Compliance", a pesar de que tiene ciertas restricciones frente a sus propias especificaciones; dispone de un subset de la implementación del estándar. MySQL provee varias extensiones de la funcionalidad para completarse.

Para la creación de una aplicación se necesitan solamente dos tipos de entidades: tablas y vistas, ambas disponibles en MySQL. Complementariamente, es útil poseer otras herramientas como procedimientos almacenados, procedimientos preparados, triggers, transacciones, constraints, y velocidad; todo esto se lo consigue mediante la combinación del lenguaje de programación PHP con el DBMS escogido.

Sin embargo, existen ciertas limitaciones en el diseño del funcionamiento del motor de la base de datos, uno de los más importantes es el tipo de tabla MyISAM ya que no posee un bloqueo de fila sino sólo un bloqueo de tabla, lo que quiere decir que al realizarse una transacción, el motor aplica un bloqueo sobre todos los registros de la tabla impidiendo una manipulación paralela de los datos; como solución se incluyó dentro del diseño de MySQL el tipo de tabla InnoDB que permita este bloqueo.

En contraposición con otros motores comerciales MySQL no posee procesamiento directo o nativo de datos XML, por tanto es necesario recurrir a lenguajes de programación para cargar información de feeds o RSS, por ejemplo, que en otras bases de datos comerciales se lo puede realizar directamente.

Bajo el ojo de la crítica, una restricción por la que MySQL solo es utilizable para aplicaciones pequeñas o personales, es que no tiene la capacidad OLAP (OnLine Analytical Processing), es decir la capacidad de gestión y análisis de datos multidimensionales, también conocida como Data Warehouse.

Adicional a la visión técnica, es preciso señalar beneficios en términos de licencias; MySQL es una base de datos libre, específicamente un proyecto Open Source, lo que implica que es de acceso libre, con la opción de adquirir una licencia comercial si se requiere.

La problemática de las licencias es compleja por la sutil diferencia entre el uso comercial de la base de datos que sí implica un alto costo, y un uso no comercial que no por ello, implica que sea sin fines de lucro.

Como alternativa MySQL existen otros gestores de bases de datos también libres e igual de flexibles, por ejemplo Postgresql, de mayor robustez en el almacenamiento de datos geoespaciales. Siendo nuestra prioridad mapear entidades de negocio, MySQL es un motor más ligero y funcional desde su instalación inicial.

Lamentablemente, la disputa entre los defensores de MySQL y Postgresql, está por debajo de los lineamientos académicos, reemplazado por un debate de posiciones y gustos[15].

2.4 Gramática

La gramática es un formalismo esencial para describir la estructura de un programa en un lenguaje de programación cualquiera. Es un principio que debe ser tomado en cuenta para definir los parámetros que determinan si un programa o fragmento de programa es válido y que podrá luego ser interpretado.

Para este trabajo nos centraremos en las gramáticas libres de contexto que permiten mayor libertad en cuanto a las reglas de producción y además no dependen del contexto para poder ser interpretadas.

En un principio una gramática define solamente la estructura sintáctica del programa. Es decir, la gramática es la receta para construir elementos mediante el uso de un conjunto de símbolos, también llamados tokens. Este conjunto de símbolos son agrupados de una manera estructurada y definida de tal manera que después puedan ser atados a una semántica particular.

[15] (Michael 2005)

Visto de una manera simple e informal, una gramática consiste en un conjunto de reglas de producción y un símbolo de inicio. Cada regla debe permitir entrelazarse con otras reglas de producción o llegar a símbolos terminales, que ya no permiten producir más reglas.

Cada regla tiene la forma mostrada en la ilustración 6:

$$expresión \rightarrow '(' \ expresión \ operador \ expresión \ ')'$$

Ilustración 6 Ejemplo de regla de producción gramatical

El lado izquierdo de la regla es el nombre de la regla y el lado derecho es una posible forma que puede tomar la regla de producción. De esta manera, el lado derecho puede ser un ***símbolo terminal***, que no permite más reglas de producción, o un símbolo ***no terminal***, que permite la concatenación de más reglas de producción.

Para el nombrado de las reglas y las partes de ella existen ciertos estándares[16]:

- No terminales son denotados con letras mayúsculas, principalmente A, B, C y N.
- Terminales son denotados con letras minúsculas cercanas al final del alfabeto, como x, y, ó z.
- Secuencias de símbolos gramaticales son nombrados con letras griegas cercanas al inicio del alfabeto: α, β, γ.
- La secuencia vacía o símbolo vacío se define con la letra épsilon: ε.

Una vez que se tienen definidas las reglas de producción y se tiene un estándar de nombrado se debe pasar al proceso de producción. La estructura de datos principal para el proceso de producción gramatical es la sentencia.

Una sentencia es un conjunto de símbolos gramaticales que puede ser una representación parcial o total de un programa producido por una gramática. Esta puede representarse mediante tokens y, además, de forma gráfica mediante árboles de producción.

Para ilustrar de mejor manera se utilizará la siguiente gramática básica[17]:

[16] (Grune Dick 2012)

1. $expresión \rightarrow '(' \; expresión \; operador \; expresión \; ')'$
2. $expresión \rightarrow '1'$
3. $operador \rightarrow '+'$
4. $operador \rightarrow '*'$

Donde los terminales son cadenas (strings) literales y los no terminales son IDs que hacen referencia a otras reglas de producción.

Entonces la sentencia:

$$(1 * (1 + 1))$$

Producida por la gramática previamente definida, se puede representar gráficamente con el árbol de producción expuesto en la ilustración 7:

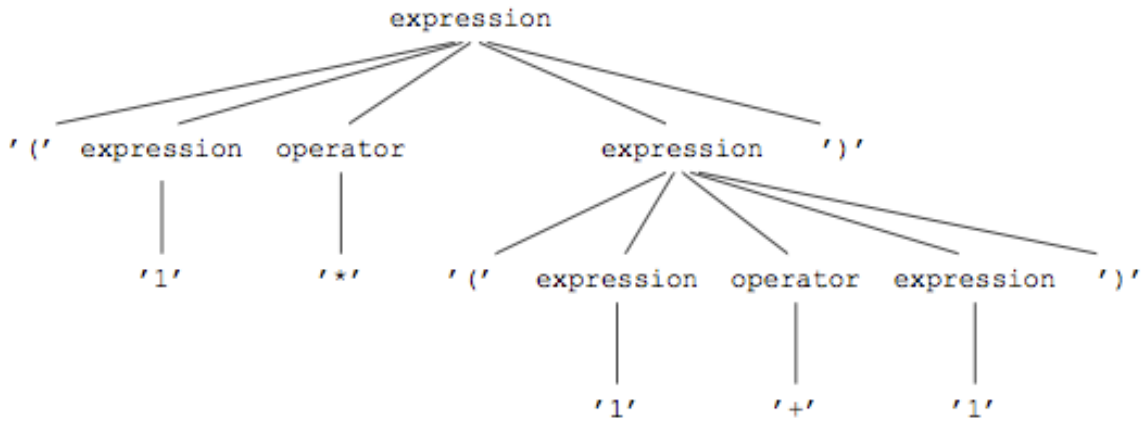

Ilustración 7 Árbol de producción de la sentencia (1 * (1 +1)) (Grune Dick 2012)

Para brindar mayor facilidad y flexibilidad en la definición de gramáticas, es permitido utilizar símbolos de control de repetición como se lo hace en expresiones regulares compatibles con Perl (PCRE). Estos símbolos son: +, ?, y * que significan: uno o más veces, cero o una vez y cero o muchas veces respectivamente, además que se puede utilizar recursión para generar estructuras de repetición más complejas.

$$expresión \rightarrow expresión \; '+' \; factor \mid factor$$ [18]

[17] (Grune Dick 2012)

[18] La regla de producción señalada genera repetición indefinida mediante el uso de recursión derecha.

Para que una gramática sea aplicable para la construcción de un fragmento de programa debe ser no ambigua. Una gramática es ambigua cuando se pueden construir dos árboles de producción con la misma sentencia yendo en el mismo orden.

La problemática de utilizar una gramática ambigua para la producción de un programa es que las reglas de producción se pierden en el proceso de producción a causa de la linearización de las sentencias, es decir, las sentencias no mantienen el orden jerárquico del proceso de producción. Y debido a que no puede ser reconstruido el árbol de producción certeramente, la semántica de la sentencia se pierde.

Finalmente, cuando se habla de una gramática G, el lenguaje producido por la misma L(G) es el conjunto de todos los programas que son correctos en ese lenguaje en un sentido libre de contexto producidos por la gramática G.

2.5 Patrones de Diseño

Lo patrones de diseño son fruto de años de experiencia en la programación; nacen como un esfuerzo de la comunidad de desarrollo para evitar errores comunes y lograr mayor robustez en el diseño y solución de problemas.

En la medida que los ingenieros de Software identificaron problemas repetitivos y persistentes, describieron diseños para lograr soluciones efectivas y reusables. La motivación es que el desarrollo se lo realizaba de manera informal y con resultados inestables; la comparación es que "si los constructores hicieran las casas de la misma forma en que los programadores escriben código, el primer pájaro carpintero que viniera destruiría la civilización"[19]

Con el tiempo, se construyó un catálogo de patrones que permitió a los novatos en el desarrollo de Software, se beneficiaran de la experiencia colectiva precedente. En 1995 nace el más famoso catálogo de patrones de diseño orientados a objetos, publicado por GoF (Gang of Four)[20], en el cual se define el lenguaje de patrones, con terminología especializada, con una estructura aceptada y ampliamente difundida por

[19] (Stelting Stephen 2003)

[20] Gang of Four es la forma coloquial de referirse a Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.

la comunidad; presentan el nombre del patrón, clasificación, su uso, beneficios y ejemplos prácticos.

Pero esta publicación (de GoF) representa la cumbre de un movimiento de desarrollo que nace hace 6 décadas. En los años 50s el desarrollo de software era guiado solamente por la experiencia personal del programador, la abstracción llegaba al nivel de algoritmos y el proceso era guiado por los resultados; es decir, por la prueba y el error.

Las décadas siguientes, 60s y 70s, fueron marcadas por el aparecimiento de las estructuras de datos o datos complejos, que, en conjunto con los algoritmos generaban soluciones más robustas. Aunque existía experiencia en el tema de las estructuras de datos y se educaba a los desarrolladores al respecto, las aplicaciones seguían siendo construidas en base a la prueba y el error en su mayor parte.

Sin embargo, empezaban a nacer las metodologías de desarrollo. Se usaban ciclos de vida como la cascada, y se empezaban a modelar los sistemas en base a flujos de datos. Entonces, en las décadas de los 80s y 90s, surge una idea que revolucionaría el desarrollo: la programación orientada a objetos.

La combinación de estructuras de datos más algoritmos como una unidad funcional motiva a las mentes creativas a desarrollar modelos que otros desarrolladores podrían seguir sin mayor complicación para resolver problemas recurrentes. Los pioneros en este tema son Ward Cunningham y Kent Beck que adaptan por primera vez la idea de "patrones de diseño" para la creación de interfaces[21].

Finalmente en 1991 Erich Gamma, uno de los autores de la pandilla de los cuatro, presenta como tesis de doctorado un paper sobre los patrones de diseño aplicados al desarrollo de software. A partir de ello, esta idea cobra fuerza e impulsados por GoF las publicaciones de patrones cada vez fueron más frecuentes.

A partir de la documentación formal de los patrones de diseño, su uso se vuelve más frecuente porque son soluciones probadas, eficaces y eficientes, por tanto el

[21] En los años 70s Christopher Alexander arquitecto de profesión escribe varios textos aplicando ideas de patrones a la arquitectura.

desarrollo se vuelve más rápido y robusto, menos dependiente del conocimiento personal del desarrollador; es decir, el código se vuelve reusable y el nivel de abstracción llega a solucionar problemas, independiente del contexto en que se use.

Una estimación del nivel de reutilización y abstracción se puede ver en la siguiente tabla:

Tipo de reutilización	Reusabilidad	Abstracción	Genericidad
Fragmento de código	Muy pobre	Nada	Muy Pobre
Estructura de datos	Buena	Tipo de datos	Moderada-buena
Funcional	Buena	Método	Moderada-buena
Plantilla	Buena	Operación para tipo	Buena
Algoritmo	Buena	Fórmula	Buena
Clase **Interfaz** **Polimorfismo** **Clase Abstracta**	Buena	Datos + métodos	Buena
Biblioteca de código	Buena	Funciones	Buena - Muy buena
API	Buena	Clases útiles	Buena - Muy buena
Componente	Buena	Grupo de Clases	Buena - Muy buena
Patrón de diseño	Excelente	Solución a problemas	Muy buena

Tabla 1 Comparación de las aproximaciones para reutilización y abstracción (Stelting Stephen 2003)

Además se forma una documentación formal sobre los pros y contras que conlleva cada solución. Por lo tanto se facilita la compresión del problema y los efectos que pueden haber al implementar una solución determinada. Asimismo, crece el vocabulario común de carácter técnico entre los desarrolladores, esto facilita la comunicación entre colegas y el grado de especificidad de la comunicación.

Los patrones de diseño son una forma fácil de compartir la experiencia obtenida, en el desarrollo, a la comunidad de una forma independiente de la tecnología que se use en el desarrollo de aplicaciones.

2.5.1 Abstract Factory

La fábrica abstracta, Abstract Factory originalmente en inglés, es un patrón de diseño orientado a objetos que provee una interfaz común para crear familias de objetos relacionados o dependientes sin la necesidad de especificar una implementación concreta[22].

Entre los beneficios de la fábrica abstracta, se encuentra el que permite crear objetos complejos de forma fácil, conveniente y sin copiar o pegar código. Es un patrón tan conveniente que se vuelve fundamental para la construcción de otros patrones más complejos, es decir, es la base para crear otros patrones de diseño.

Otra ventaja de usar este patrón es el "lazy loading" de los recursos de la aplicación. Este concepto se refiere a que los objetos se cargan solamente cuando es estrictamente necesario y, por lo tanto, no se desperdician recursos a lo largo de la ejecución del programa.

Pero, sin duda, el beneficio más grande de la fábrica abstracta es el polimorfismo y la encapsulación que genera. Por un lado retorna objetos de diferentes clases usando una misma interfaz común y por otro lado permite encapsular el proceso de creación de los objetos de una misma familia.

Gracias a ésto, el método fábrica abstracta solamente retorna la instancia necesaria sin descubrir a que clase concreta corresponde y además si en algún caso se necesita modificar la estructura de creación o el tipo de objeto utilizado simplemente se cambia el método fábrica y el resto de la aplicación estará actualizada al instante.

Sin embargo, posee una desventaja que debe ser tomada en consideración, los puntos de evolución de la familia de clases que implementen la fábrica abstracta son varios. En otras palabras, cuando la aplicación es ampliada en funcionalidad se necesita implementar varios pasos previos a la funcionalidad en sí misma.

Se debe tomar en cuenta que una nueva implementación requeriría: una nueva clase abstracta, una nueva clase concreta que implemente la abstracta, la herencia de la interfaz abstracta y clases derivadas que implementen extensiones. Es una serie de

[22] (Gamma Erich 1994)

pasos que fácilmente puede confundir a un desarrollador no familiarizado, pero al fin siempre se beneficia la reutilización y la robustez de la aplicación[23].

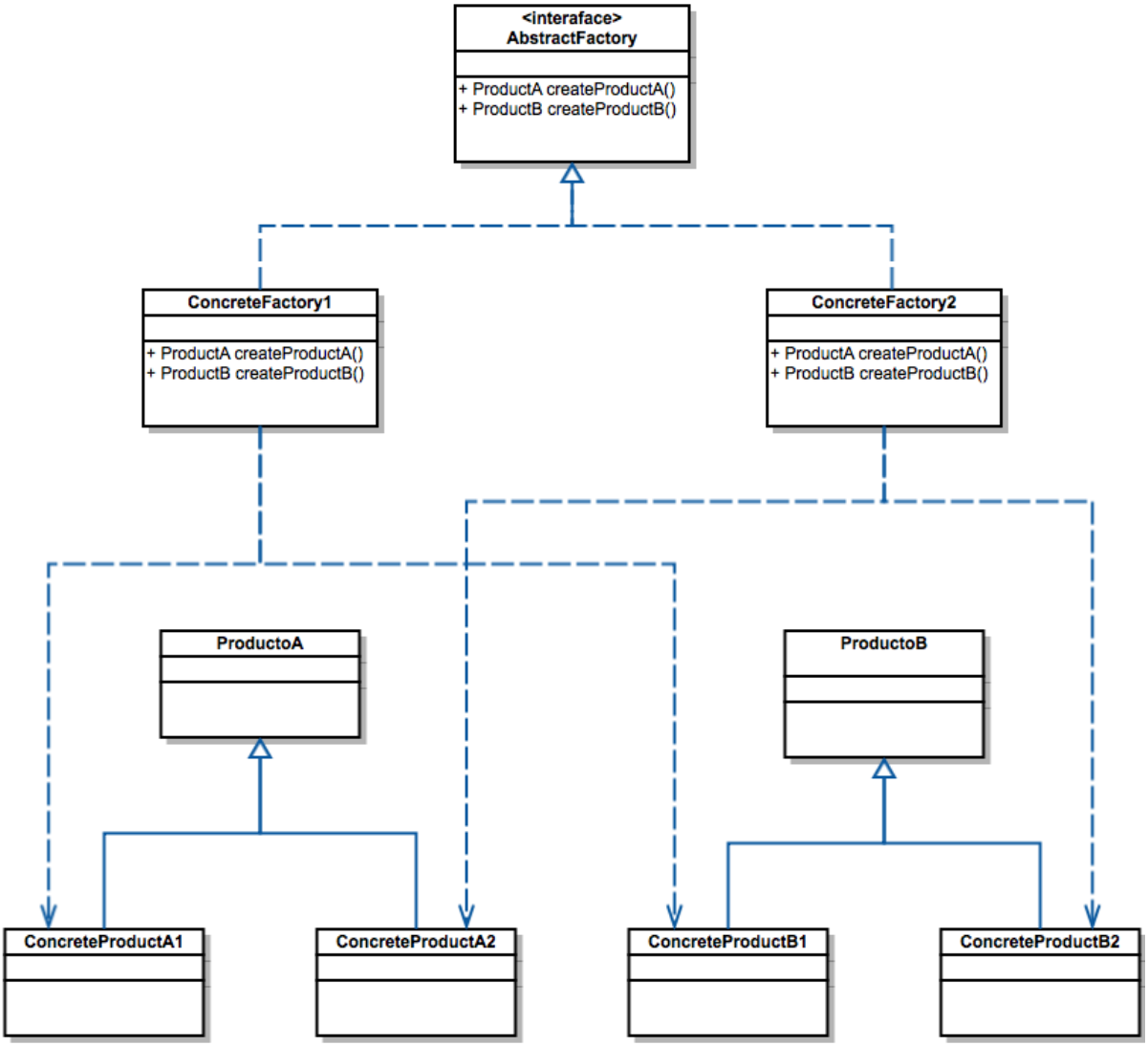

Ilustración 8 Diagrama de Clases de Abstract Factory (Stelting Stephen 2003)

El diagrama de clases general para la fábrica abstracta es sencillo pero muy robusto y extensible, tal como se muestra en la ilustración 8.

Este patrón de diseño cumple una función importante en el presente trabajo porque a partir de él se generará la librería de generación de clases en PHP. Haciendo uso de métodos fábrica se podrá generar fácilmente la serie de componentes que se necesitan para poder crear una máquina generadora de código automatizada.

[23] (Kulkarni 2013)

Además provee una base lo suficientemente flexible para que en un futuro se pueda extender la funcionalidad de la librería para desarrollar un producto más completo. Se podrá agregar en futuros desarrollos otros procesos para generar un ORM viable comercialmente.

CAPÍTULO 3 METODOLOGÍA DE DESARROLLO

En este capítulo se desarrollan los conceptos sobre la metodología de desarrollo que será utilizada para la elaboración de la librería objetivo de este trabajo. Se mencionan los fundamentos de la metodología, su ciclo de vida, las medidas que utiliza y la manera de hacer el seguimiento del plan durante la creación del producto.

3.1 PSP (Personal Software Process)

El PSP es una metodología de desarrollo de software propuesta por Watts Humphrey. Esta metodología fue diseñada para ayudar a los ingenieros a realizar un mejor trabajo, porque ayuda a los desarrolladores a controlar su eficiencia, a seguir adecuadamente los planes de los proyectos, es decir, a aplicar métodos de ingeniería de software en sus tareas diarias.

Esta metodología ha sido utilizada en numerosas universidades e industrias a nivel global: "Los datos reunidos de los cursos impartidos muestran que el PSP es efectivo a la hora de mejorar el rendimiento en la planificación de los ingenieros y la calidad de sus productos"[24].

En casos exitosos de aplicación del PSP se ha llegado a mejorar el tiempo de desarrollo de los aplicativos en 10.4% respecto a los tiempos estimados en la planificación. Además en comparación con proyectos elaborados sin metodología se consiguen productos cinco veces superiores en calidad, en otras palabras, se reduce el números de defectos en un 80%[25].

Seguir una metodología no es fácil y PSP no es una excepción. El trabajo necesario para cumplir con los métodos y técnicas es considerable y requiere madurez en el ejercicio de la ingeniería.

La meta de aplicar PSP es fortalecer la capacidad propia y a medida que pasa el tiempo reducir la cantidad de trabajo que se debe invertir en mantenimiento correctivo de los proyectos. Para lograrlo se debe aprender de los errores cometidos en proyectos

[24] (Humphrey 2001)
[25] Ibíd.

anteriores, esto se consigue mediante el análisis de los datos que han sido recolectados durante el desarrollo.

Humphrey diseñó esta metodología con el objetivo de que sea aplicada en proyectos de desarrollo de software en los que se debe trabajar individualmente. Por lo tanto resulta muy adecuada para la elaboración de este proyecto.

3.2 Conceptos Básicos

Cuando se elabora un proyecto a nivel profesional se necesita cumplir altos estándares de calidad, para ello es necesario que el ingeniero de software utilice los recursos planificados sin sobrepasarlos[26]. Para ello se debe realizar un trabajo efectivo que se caracteriza por productos de calidad, costos dentro de lo esperado y el cumplimiento de plazos establecidos.

Muchos ingenieros, a través del fracaso de sus proyectos, se han dado cuenta que se necesita algo más que el conocimiento para conseguir un trabajo efectivo. Según Watts Humphrey[27]:

- Planificar el trabajo
- Hacer el trabajo de acuerdo al plan
- Esforzarse en elaborar productos de calidad

La importancia de una buena ingeniería se ve en las conductas comerciales, si una organización no cumple las metas fijadas con el cliente se puede dañar su reputación inclusive al punto de perder el proyecto. Esto toma mayor relevancia debido a que actualmente el software es crítico para las industrias, la tecnología maneja desde transferencias de dinero por sumas millonarias hasta monitoreo médico en tiempo real para pacientes en estado grave.

Por todo esto el software debe ser cada vez más efectivo y eficiente. Por lo tanto los grupos de ingenieros tienen que serlo también y la única forma de lograrlo es mediante metodologías. No obstante, generar un cambio es complicado porque los ingenieros están acostumbrados a trabajar a su manera.

[26] Recursos se refiere a tiempo, dinero y personal.
[27] (Humphrey 2001)

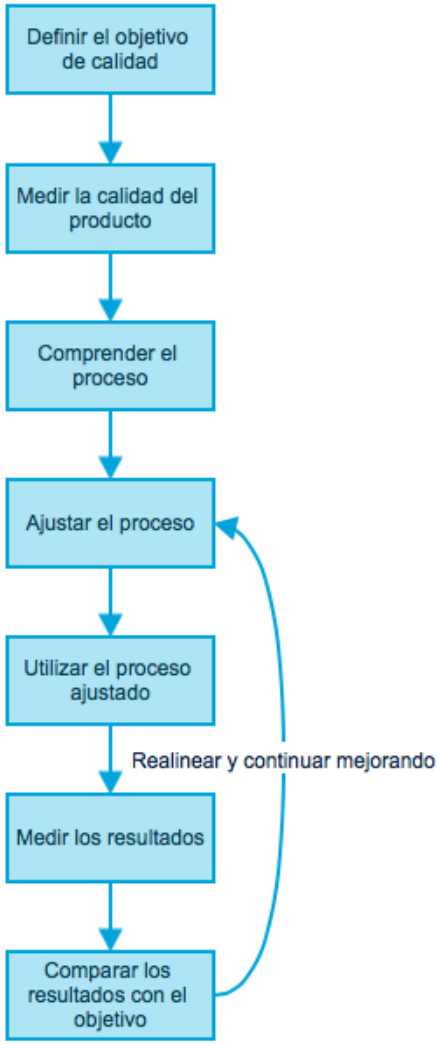

Ilustración 9 El proceso de mejora (Humphrey 2001)

Tal como se muestra en la ilustración 9, para poder mejorar el trabajo se debe primero observar cómo se lo hace, medir objetivamente el proceso y finalmente se encontrará donde existe posibilidad de mejora. Sin embargo definir medidas adecuadas para el software no es una tarea sencilla y, aunque se encuentre la medida, el paso más difícil es realizar el cambio en sí mismo.

3.3 Ciclo de Vida

El ciclo de vida de PSP asume que ya se dan los requerimientos listos, a partir de ello se tiene lo siguiente:

3.3.1 Planificación

Existen dos tipos de planificaciones, que pueden variar en nivel de detalle utilizado: basada en un período de tiempo, es decir se organiza la forma de utilizar el tiempo

durante el periodo. O, la otra forma, basada en actividades, como cuando escribes un informe o un programa, donde se detallan actividades organizadas jerárquicamente hasta cumplir con todo el trabajo.

Ambas formas de planificación están estrechamente relacionadas. A pesar de ello se detallará más a profundidad la planificación guiada por el producto, es decir, basada en actividades.

La planificación del producto es muy importante porque indica límites de cuándo será finalizado el producto y los costos de elaboración del mismo. Esto quiere decir que si el plan es inexacto, entonces los períodos de tiempo y costos estimados serán también imprecisos, esa es la importancia de realizar planes adecuados.

Además, es importante cumplir con las metas fijadas para el proyecto, como dice Watts Humphrey hablando sobre un grupo de ingenieros de IBM: "El acto de hacer planes tiene un efecto sorprendente: este grupo de desarrollo nunca había entregado anteriormente un producto a tiempo. Con sus nuevos planes, no se equivocaron en una fecha de entrega durante los dos años y medio siguientes"[28].

La planificación es la base fundamental que se necesita para poder comprometerse con las fechas de entrega. Los planes ayudan a entender el estado del proyecto, el avance que tiene. Si un plan es moderadamente detallado, preciso y confiable, entonces ayuda a juzgar el avance del proyecto de manera real.

Para realizar un buen plan lo primero es saber exactamente qué se quiere producir, en otras palabras, se debe tener definido el producto. A partir de ello se puede empezar a planificar cómo hacerlo, para esto se necesita[29]:

- El tamaño y las características más importantes del producto a realizar.
- Una estimación del tiempo requerido para hacer el trabajo.
- Una previsión de la planificación.

[28] (Humphrey 2001)
[29] Ibíd.

3.3.2 Diseño

El diseño en un término con una definición vaga y poco clara, puede ser cualquier cosa que se relacione con la estructura e implementación de un programa, desde un flujo de datos hasta una descripción de la naturaleza de los elementos del lenguaje que será utilizado. Lo único que se utiliza como guía para definir al diseño es el nivel de detalle que se utilice.

Precisamente por esto, la problemática del diseño se toma desde dos perspectivas: el diseño a alto nivel (componentes, arquitectura) y el diseño a detalle (estructuras de repetición, cases, clases).

En este punto es posible darse cuenta que la separación entre el diseño y la implementación es arbitraria, cada ingeniero de software decide hasta que punto "diseñar" y desde que punto se le llama implementación. A nivel general se suele manejar el diseño desde las dos perspectivas antes mencionadas y esto se debe a que esta separación durante el proceso de diseño, permite enfocarse de mejor manera en las cuestiones adecuadas en el momento correcto.

El esfuerzo de diseñar a diferentes niveles de detalle tiene por objetivo que en la fase de implementación se cometan menos errores. En un nivel alto de abstracción los diseñadores pueden modelar toda la estructura lógica de un sistema sin preocuparse de los detalles sucios de la implementación y posteriormente se llega a un diseño detallado que permite la transición hacia la fase de implementación sin ambigüedades y evitando errores.

Durante el proceso de diseño los ingenieros de software más experimentados normalmente se mueven entre niveles de abstracción. Esto se debe a que es complejo llegar a abstraer a alto nivel mecanismos que no se conoce como funcionan.

Entonces, al menos que sean componentes reutilizados o frecuentes es posible que se deba llegar al detalle antes de dar por terminado el diseño a alto nivel. Si el diseño, aparentemente sencillo, no es comprobado a otros niveles de detalle antes de concluirlo puede suceder que durante la implementación se descubra que no es viable o no cubre todas los requerimientos del programa.

Si es que se utiliza este principio, mezclar niveles de abstracción para comprobar el diseño, se puede ver claramente que la separación entre diseño e implementación depende totalmente del ingeniero de software. Por lo tanto, para fines de este proyecto, llamaremos diseño a lo que se haga durante la fase de diseño, aunque esto parezca "tan útil como definir *pensamiento* como lo que la gente hace cuando piensa"[30].

Dejando de lado la problemática de la definición del diseño hay que mencionar también que, si se tiene un buen diseño que sea claro y fácil de interpretar para cualquiera, entonces se introducirán menos defectos durante la implementación. Lo que nos introduce la incógnita de cómo representar el diseño de forma clara y no ambigua.

Es más probable que se construya un producto de calidad si es que se tiene un diseño completo y fácil de entender, para conseguirlo se utilizan normalmente 3 técnicas de representación: gráficos, pseudocódigo y notación matemática.

A pesar de que se pueden conseguir buenos diseños con cualquiera de las técnicas mencionadas, este proyecto será modelado utilizando herramientas gráficas. Se utilizará un lenguaje especialmente diseñado con este objetivo: el UML.

El lenguaje unificado de modelado (UML) permite representar el diseño de un programa mediante gráficos claros, precisos y no ambiguos, eliminando así los problemas de interpretación del diseño.

Finalmente, un paso previo a la implementación que debe ser definido en el diseño son los estándares de codificación que serán utilizados. "Un estándar de codificación define un conjunto de prácticas de codificación aceptadas, las cuales pueden servir como un modelo para tu trabajo"[31].

En los estándares normalmente se menciona cómo va a estar formateado el código, que líneas irán separadas, formato para comentarios, etc. Y además proveen el

[30] (Humphrey 2001)
[31] Ibíd.

beneficio de que evitan errores de sintaxis y facilita realizar revisiones de código para evitar errores lógicos, porque el código estará más limpio y ordenado.

3.3.3 Implementación

La implementación es la fase en la que el producto toma forma. A través de la codificación el programa adquiere su funcionalidad y se ve en este momento el resultado de las anteriores fases antes descritas.

En este paso de la metodología se deben utilizar los productos de la planificación y diseño y seguirlos al pie de la letra para poder generar un producto de calidad y que cumpla con los requerimientos necesarios. Para ello es necesario tener completamente descrito cómo cumplir con todas las tareas o actividades que el PSP requiere.

Por este motivo existe un "guión"[32] del proceso, que describe paso a paso qué se debe realizar y en qué orden para llegar al producto final. En la tabla mostrada más abajo se presenta el guión para el desarrollo de este proyecto, sin embargo, cabe recalcar que es una versión ligeramente modificada del guión estándar de PSP; los cambios introducidos tienen el fin de ajustar el guión a las necesidades propias del producto.

Al final de cada fase del proyecto se generan productos entregables de la fase. Estos productos son útiles para medir el nivel de avance del proyecto. Estas mediciones se realizan en los puntos de control que, para finalidad de este proyecto, coincidirán con el fin de cada fase.

[32] Cuando un proceso está completamente descrito, se lo llama "proceso definido" y generalmente poseen "guiones".

	Propósito	Guiar en el desarrollo de programas
	Criterios de Entrada	- La descripción del problema.
1	**Planificación**	- Descripción de las funciones principales del programa. - Estimación de LOC Max., Min. - Calcula tiempos de desarrollo Max., Min. - Escribe los datos del plan. - Hacer seguimiento del tiempo utilizado.
2	**Diseño**	- Diseña el programa. - Especificar el diseño en UML. - Hacer seguimiento del tiempo utilizado.
3	**Codificación**	- Implementa el diseño. - Utilizar estándares de codificación especificados en el diseño. - Corregir errores de tiempo de ejecución. - Hacer seguimiento del tiempo utilizado.
4	**Pruebas**	- Prueba el programa. - Corregir errores encontrados. - Hacer seguimiento del tiempo utilizado.
5	**Postmortem**	- Generar análisis de tiempos y tamaños reales vs. estimados. - Hacer seguimiento del tiempo utilizado.
	Criterios de Salida	- Programa probado a fondo. - Diseño adecuadamente documentado. - Registro de tiempos completado.

Tabla 2 Guión de desarrollo de PSP para el proyecto

3.3.4 Pruebas

No existe ningún método completamente efectivo para eliminar la introducción de defectos, pero si se puede eliminar casi todos desde el principio del desarrollo del programa. Si se elimina los defectos desde el principio se ahorra tiempo, trabajo y los productos serán de mayor calidad.

Para eliminar defectos se sigue una serie de pasos que todo desarrollador conoce, aunque sea intuitivamente[33]:

[33] (Humphrey 2001)

1. Identificar los síntomas del defecto.
2. Deducir de estos síntomas la localización del defecto.
3. Entender lo que es erróneo del programa.
4. Decidir cómo corregir el defecto.
5. Hacer la corrección.
6. Verificar que el arreglo ha resuelto el problema.

Para facilitar este proceso el intérprete indica varios defectos automáticamente, generalmente errores de sintaxis o de tipos de datos, conversiones u operaciones no permitidas. Sin embargo, el intérprete solamente revela síntomas de errores, es el trabajo del desarrollador entender y corregir los problemas.

Y a pesar de que el intérprete puede ayudar a encontrar los errores sintácticos solo mediante pruebas se puede encontrar los problemas lógicos del programa. Y por este motivo las pruebas son necesarias, aunque requieren bastante tiempo para ser ejecutadas.

Habitualmente se representan las pruebas como casos de prueba: consisten en señalar ciertas condiciones de entrada y el resultado esperado, para luego comparar con el resultado real arrojado por el programa; y si es que no coinciden ambas, entonces es señal de un error. Aunque parece fácil, tiene varias desventajas[34]:

- Solo muestran síntomas, todavía hay que analizar cuál es la causa del problema.
- Cada prueba verifica solo un número determinado de condiciones.
- Probar todas las condiciones del programa es casi imposible.

Entonces aún cuando hagamos pruebas, eliminar todos los defectos es imposible y además hay que tomar en cuenta que el proceso de probar el programa es lento y costoso; “durante un año, IBM gastó unos 250 millones de dólares en reparar y reinstalar correcciones de los 13000 defectos detectados por los clientes. Esto supone unos 20000 dólares por defecto”[35].

[34] (Humphrey 2001)
[35] Ibíd.

Por lo tanto hay que recurrir a métodos más eficientes de detección de defectos: la revisión de código. Aunque pueda parecer difícil y largo "los datos de estudiantes e ingenieros muestran que, la revisión de código es entre 3 y 5 veces más eficiente que ejecutar las pruebas de unidad. Un ingeniero, por ejemplo, encontrará solamente entre 2 a 4 defectos en una hora de pruebas, pero encontrará de 6 a 10 defectos en cada hora de revisión de código"[36].

Sumado a todo esto es importante notar que aunque al final del desarrollo se logre corregir muchos problemas el costo que esto llevará es mucho más alto que si se los identifica en tiempo de codificación. El costo de corregir un defecto se incrementa aproximadamente en 10 veces con cada fase que se avanza, es decir, un error que en la etapa de codificación se demora 1 o 2 minutos en la fase de pruebas puede demorarse de 10 a 20.

3.3.5 Postmortem

Una vez que se ha terminado el desarrollo y se ha probado que el programa funciona de acuerdo a lo esperado, se debe realizar un análisis de cómo se desarrolló el proyecto. Se debe definir si las estimaciones de tiempo y tamaño fueron adecuadas o en cuánto se desviaron.

Es útil presentar estos resultados en cuadros comparativos y acompañarlos de un análisis personal de la ejecución del proyecto. Esta información será de gran utilidad para futuros proyectos que serán planificados de acuerdo a los datos de proyectos pasados.

3.4 Medición y Estimación de Tamaño

El proceso de planificación es inexacto, incluso los mejores planes pueden no ser totalmente precisos. Parte del proceso de planificación es la estimación del tamaño del producto que se va a realizar, que es por lo tanto impreciso.

La habilidad de estimar adecuadamente puede desarrollarse con el tiempo y la práctica pero nunca va a ser perfecto. La clave para definir estimaciones buenas es comparar lo que se va a hacer con el trabajo que se ha realizado en el pasado.

[36] (Humphrey 2001)

Justamente por este motivo es que se deben recolectar datos para generar una base de conocimiento que servirá en el futuro.

En apariencia es un proceso sencillo pero no lo es, se deben tomar en cuenta que los diferentes tipos de trabajo requieren diferentes estimaciones de tiempo. Existen trabajos más sencillos que requerirán menos tiempo que otros más complejos, aunque el tamaño sea similar.

Además el tiempo utilizado depende mucho de la intención que se tenga. Por ejemplo, un estudio concienzudo requiere mucho más tiempo que un repaso rápido aunque el objeto sea el mismo documento.

La estimación se complica un poco más al momento de escoger una medida que permita comparar diferentes trabajos. Por ejemplo, para estimar tiempos de lectura sería una buena idea medir páginas por minuto.

Para este trabajo se estimarán se definirán tres medidas:

- Tamaño de desarrollo teórico
    - Tamaño (hojas)
- Tiempo (horas)
- Tamaño del producto
    - Tamaño (líneas de código)

Las estimaciones del tiempo se deben basar en el tamaño que se espera que tenga el producto y la unidad depende del tipo de trabajo que se realice. Para la documentación se medirá el tamaño en páginas y para la librería se utilizarán líneas de código (LOC).

El tamaño del programa podría ser medido en archivos o clases, pero para poder hacerlo comparable con otro tipo de desarrollos el conteo de líneas de código es la mejor medida. Para el conteo de líneas se asume que no existen líneas vacías ni comentarios y que la codificación se realiza a un ritmo constante manteniendo coherencia en el estándar.

3.5 Seguimiento del Plan

"Probablemente harás esta semana lo mismo que hiciste la semana pasada. En general, la forma en que utilizaste tu tiempo la última semana te proporcionará una aproximación bastante buena a la forma en la que gastarás tu tiempo en futuras semanas"[37]. Por este motivo, para poder hacer planes realistas se debe efectuar un seguimiento de cómo se gasta el tiempo en un proyecto.

Existen dos aspecto interesantes que se deben tomar en cuenta cuando se realiza un proyecto: en general el tiempo que se gasta en realizar una tarea o trabajo es menor que el que se estima, es decir, menor al que se cree que se gasta. Por otro lado, muchos de los problemas que surgen durante un proyecto son causados porque los ingenieros realizan su trabajo llenos de distracciones y sin reflexión.

Por eso es esencial tener un plan detallado y sobretodo seguirlo al pie de la letra. Con el seguimiento estricto de un plan es menos probable que se desperdicie tiempo y que se cometan errores. Para poder tener un plan adecuado y seguirlo rigurosamente se debe:

1. Clasificar las principales actividades.
2. Registrar el tiempo dedicado a cada una de las actividades principales.
3. Registrar el tiempo de forma normalizada.
4. Guardar los datos de tiempo en un lugar adecuado.

El seguimiento de un plan de forma estricta incluye el registro de tiempo que se demora una tarea, pero también se refiere a la gestión de las interrupciones. Esta información es importante para poder determinar cuánto tiempo neto de trabajo se utiliza para cada tarea y además como herramienta personal para analizar el comportamiento que se tiene durante las actividades.

Por supuesto este trabajo de registrar tiempo utilizado, interrupciones, tareas finalizadas, etc. no es sencillo y por este motivo se recurre a herramientas de terceros adaptadas para este objetivo, por ejemplo el Process Dashboard.

[37] (Humphrey 2001)

3.5.1 Dashboard

Para realizar un seguimiento del plan al pie de la letra recolectando las métricas necesarias para la metodología se utilizará la aplicación “Process Dashboard”, un utilitario de acceso libre hosteado en sourceforge.net que cumple con todas las especificaciones del PSP.

El process dashboard es una herramienta que permite a individuos o equipos que desarrollan software utilizando metodologías de alta madurez centradas en métricas intensivas, como PSP o TSP. Soporta el seguimiento del valor ganado (earned value en inglés) y definición de estimaciones; su interfaz se muestra en la ilustración siguiente:

Ilustración 10 Interfaz de utilitario "Process Dashboard"

CAPÍTULO 4 DESARROLLO Y PRUEBAS DEL PRODUCTO

En este capítulo se desarrollará la librería como tal y se presentará la documentación formal del producto: planificación, diseño a alto nivel, diseño a detalle; además se realizarán las pruebas del software desarrollado y se presentarán sus resultados. Finalmente se concluye con un análisis de los resultados del proyecto, incluyendo una comparación de los tiempo y tamaños estimados versus los reales.

4.1 Planificación

La librería para generación automática de clases debe consistir en un conjunto de clases que permitan al usuario enviar parámetros de conexión a un determinado esquema de la base de datos en MySQL más ciertos parámetros opcionales y la librería generará archivos .php con las clases correspondientes.

Para comprender mejor la funcionalidad requerida para la librería se ilustra su funcionalidad en los siguientes diagramas:

Ilustración 11 Flujo de datos nivel 1

Si ampliamos la funcionalidad del paso intermedio de la ilustración de arriba obtenemos:

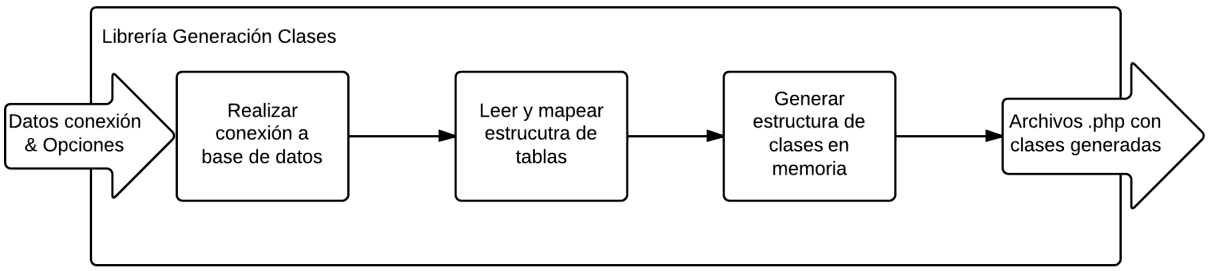

Ilustración 12 Flujo de datos nivel 2

La funcionalidad traducida a una representación gráfica típica de metodologías orientadas a objetos se muestra a continuación:

Casos de uso: General

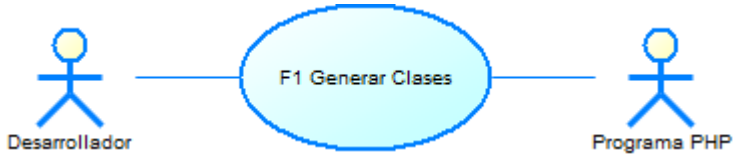

Ilustración 13 Diagrama de casos de uso general

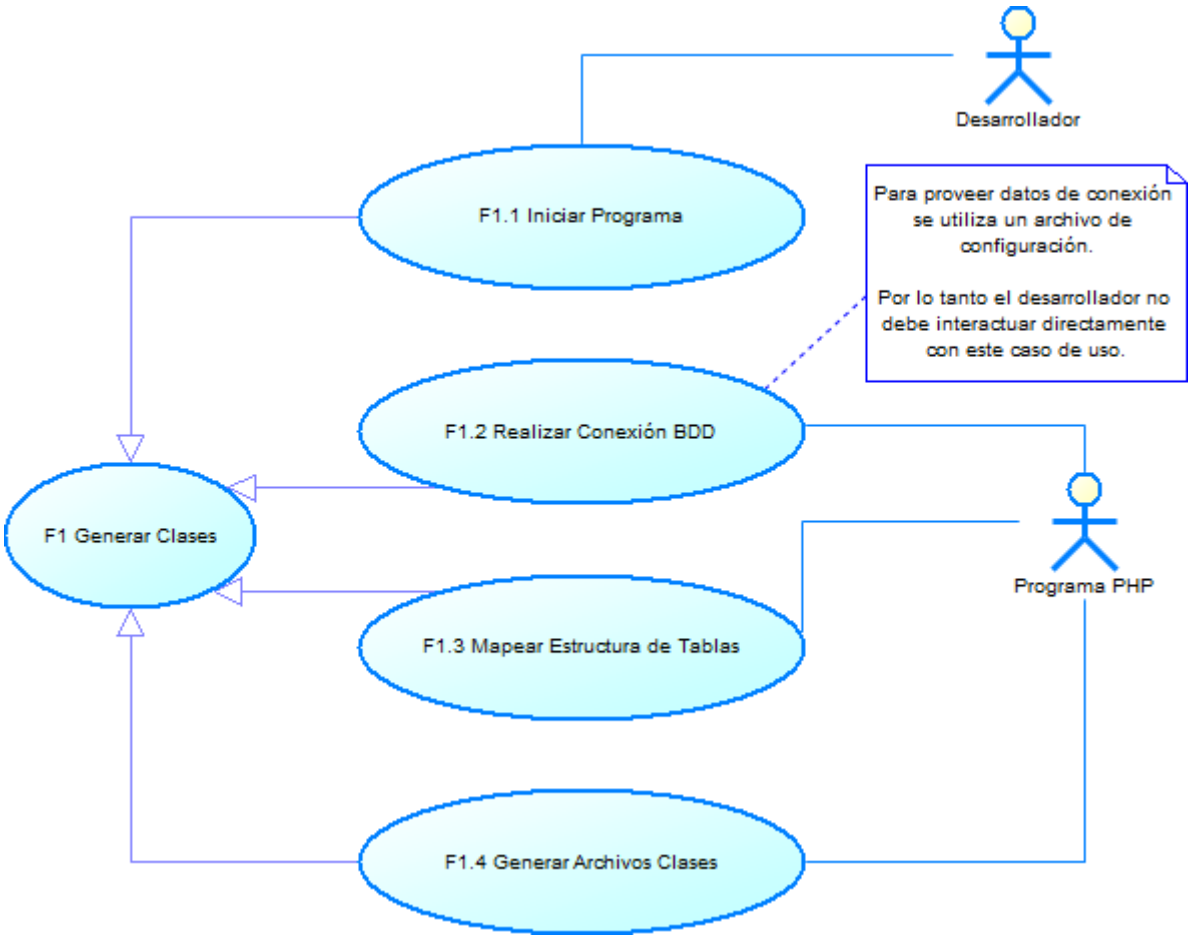

Ilustración 14 Diagrama de casos de uso segundo nivel

Casos de uso: Detalle

Casos de uso F1.1 Iniciar Programa

Ilustración 15 Diagrama de casos de uso detalle: F1.1 Iniciar Programa

Definición: Mediante este caso de uso el actor puede iniciar la ejecución de la librería para producir las clases PHP.

Actores: Desarrollador.

Precondiciones: N/A.

Flujo Principal:

1. Situarse en el directorio donde se encuentre la librería.
2. Insertar el comando para ejecutar la librería.
3. Agregar parámetros opcionales.
4. Ejecutar comando.
5. Inicia la ejecución del programa.

Flujo alterno:

5. Archivo de configuración con parámetros de conexión no existe o está incompleto (E1).

Excepciones:

E1. Se muestra mensaje de error: “Error en archivo de configuración”.

Casos de uso F1.2 Realizar Conexión BDD

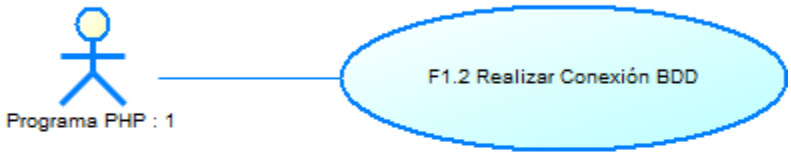

Ilustración 16 Diagrama de casos de uso detalle: F1.2 Realizar Conexión BDD

Definición: Mediante este caso de uso el actor se conecta a al BDD.

Actores: Programa PHP.

Precondiciones: Archivo de configuración con parámetros de conexión a BDD.

Flujo Principal:

1. Se leen parámetros de conexión del archivo de configuración.
2. Se intenta conectar a la base de datos mediante el driver adecuado.
3. La conexión es realizada exitosamente.

Flujo alterno:

3. La conexión falla (E1).

Excepciones:

E1. Se muestra mensaje de error: “La conexión no se pudo realizar. Revisar parámetros de conexión”.

Casos de uso F1.3 Mapear Estructura de Tablas

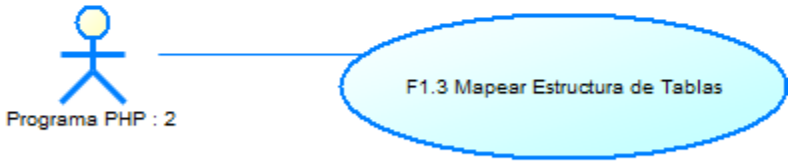

Ilustración 17 Diagrama de casos de uso detalle: F1.3 Mapear Estructura de Tablas

Definición: Mediante este caso de uso el actor mapea la estructura de las tablas de la BDD a estructuras en memoria del programa.

Actores: Programa PHP.

Precondiciones: Conexión a BDD establecida.

Flujo Principal:

1. Por cada tabla: Leer estructura.
2. Separar tokens de estructura.
3. Almacenar tokens como datos complejos en memoria.

Flujo alterno:

3. Estructura leída de BDD no compatible con datos complejos definidos en memoria (E1).

Excepciones:

E1. Se muestra mensaje de error: “No es posible mapear tabla ---”. Continuar con la ejecución normal.

Casos de uso F1.4 Generar Archivos Clases

Ilustración 18 Diagrama casos de uso detalle: F1.4 Generar Archivos Clases

Definición: Mediante este caso de uso el actor genera archivos .php que contienen las clases PHP correspondientes a las tablas de la BDD.

Actores: Programa PHP.

Precondiciones: Estructuras de BDD mapeadas como datos complejos en memoria.

Flujo Principal:

1. Por cada dato complejo de memoria: Tomar los tokens y conectarlos siguiendo reglas sintácticas específicas (E1).
2. Persistir los tokens conectados en archivos planos con extensión .php (E2).

Flujo alterno:

1. No se pueden persistir los datos complejos.

Excepciones:

E1. Tokens no se pueden conectar, no cumplen requisitos. Mensaje de error: "Error al generar estructura de clase ---".

E2. No se puede escribir el directorio de salida. Mensaje de error: "No se puede escribir el directorio de salida".

Estimación de tamaño y tiempo

La librería en sí misma debe ir acompañada de la documentación correspondiente, tanto del diseño como la referencia de funcionalidad de la misma (resumen de los métodos existentes para la manipulación de la información). Las estimaciones de tiempo y tamaño para ambos productos se muestra en la tabla siguiente:

	Estimación
Tiempo (horas)	40
Tamaño (hojas)	50
Tamaño (LOC)	1200

Tabla 3 Estimación de tamaño sin márgenes de error

Los valores definidos en la tabla anterior se calculan tomando en cuenta: los requisitos del programa, su clasificación relativa con respecto al tamaño de otros programas desarrollados y la experiencia. Sin embargo, la construcción de un producto como el descrito en este trabajo es un proceso caótico y por lo tanto se define un margen de equivocación en las estimaciones anteriores[38] tal como se expone en la tabla siguiente:

	Estimación	Error Esperado	Rango Estimado
Tiempo (horas)	40	10%	36 - 44
Tamaño (hojas)	50	5%	48 - 53
Tamaño (LOC)	1200	5%	1140 - 1260

Tabla 4 Estimación de tamaño con márgenes de error

4.2 Diseño

Una vez que se han definido los requerimientos del programa, se debe especificar la estructura de la aplicación. A nivel conceptual esto se logra mediante el diagrama conceptual de clases, para este proyecto es el expuesto a continuación:

[38] Las horas estimadas son horas de trabajo neto.

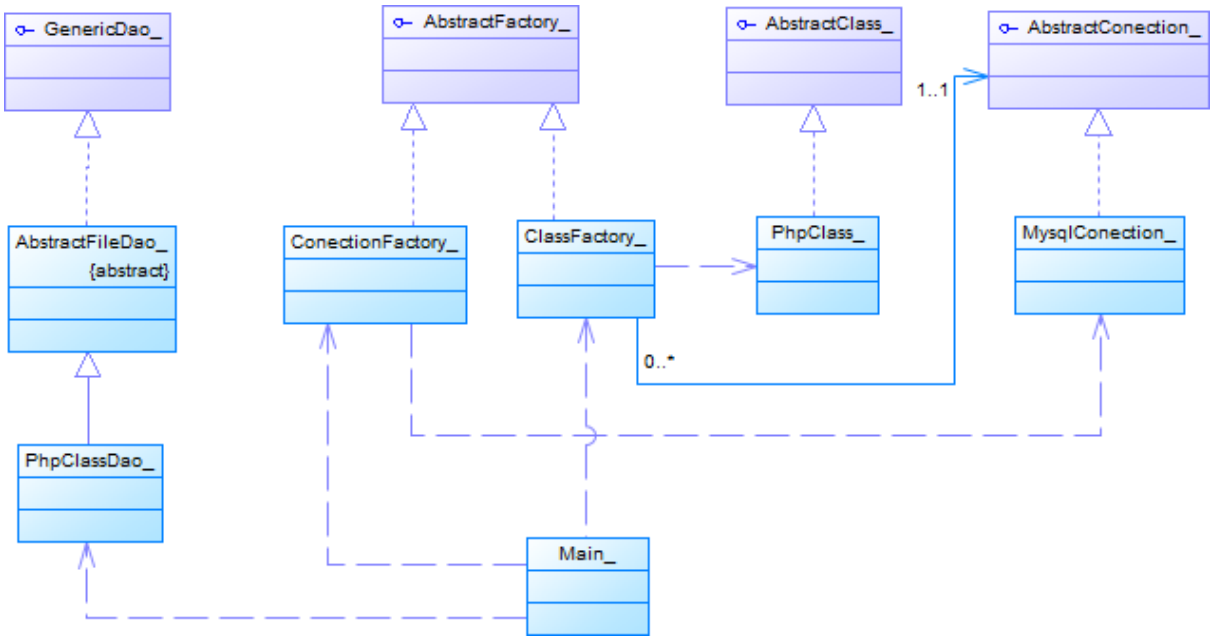

Ilustración 19 Diagrama conceptual de clases

Las clases van a estar organizadas en paquetes de acuerdo al siguiente gráfico:

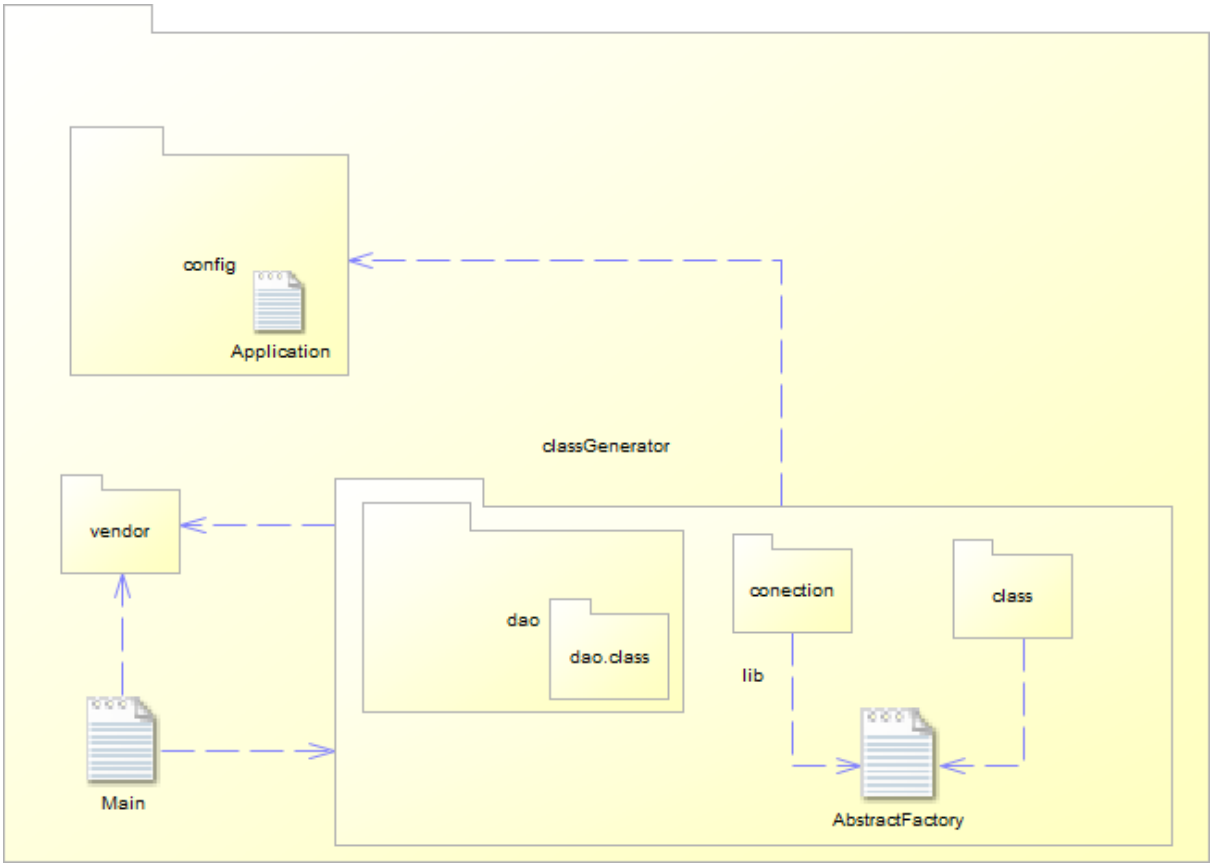

Ilustración 20 Diagrama de paquetes

Las clases a nivel detallado van a estar constituidas de la siguiente manera:

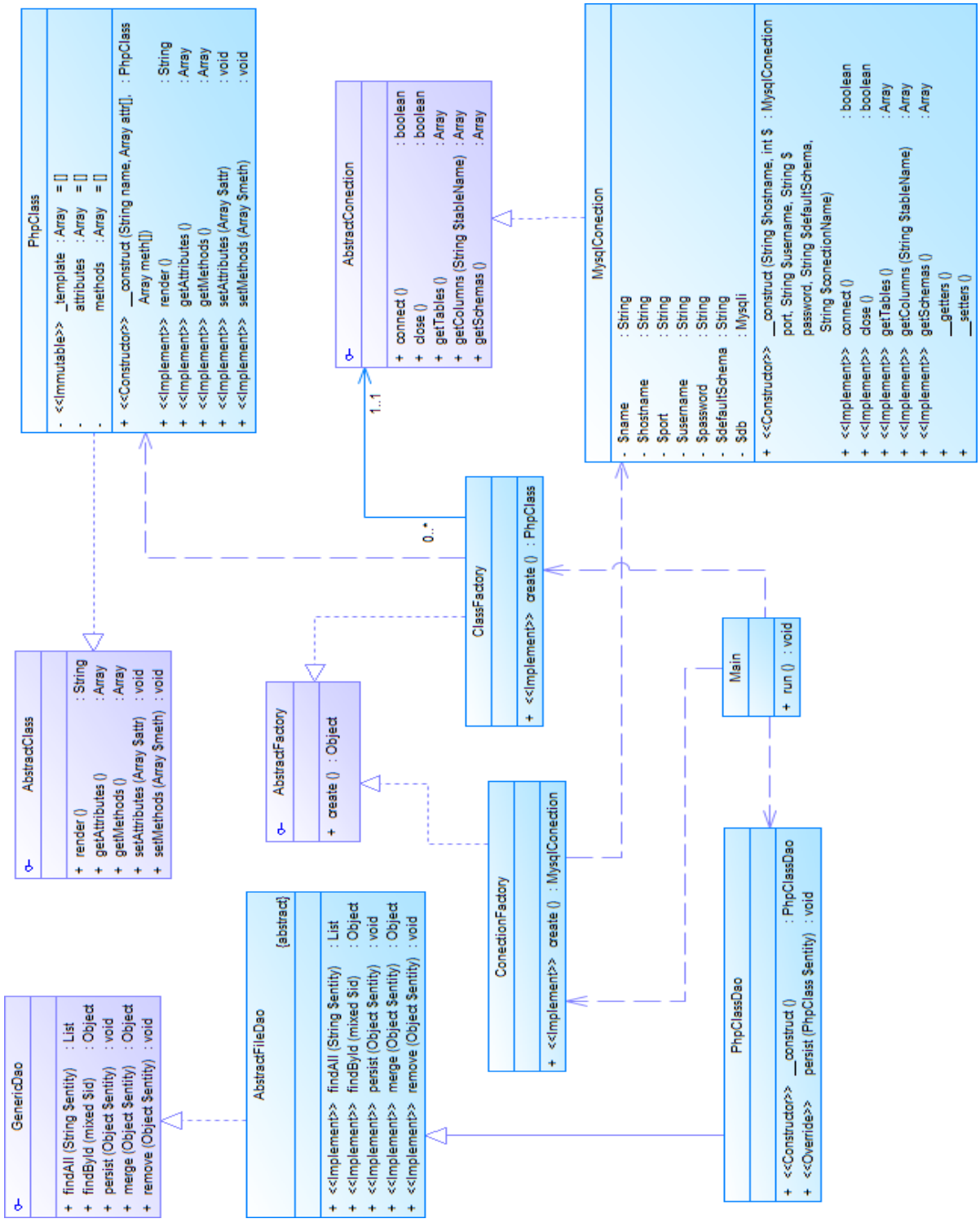

Ilustración 21 Diagrama de clases a detalle

La librería puede pasar por los siguientes estados, como se detalla en el diagrama siguiente:

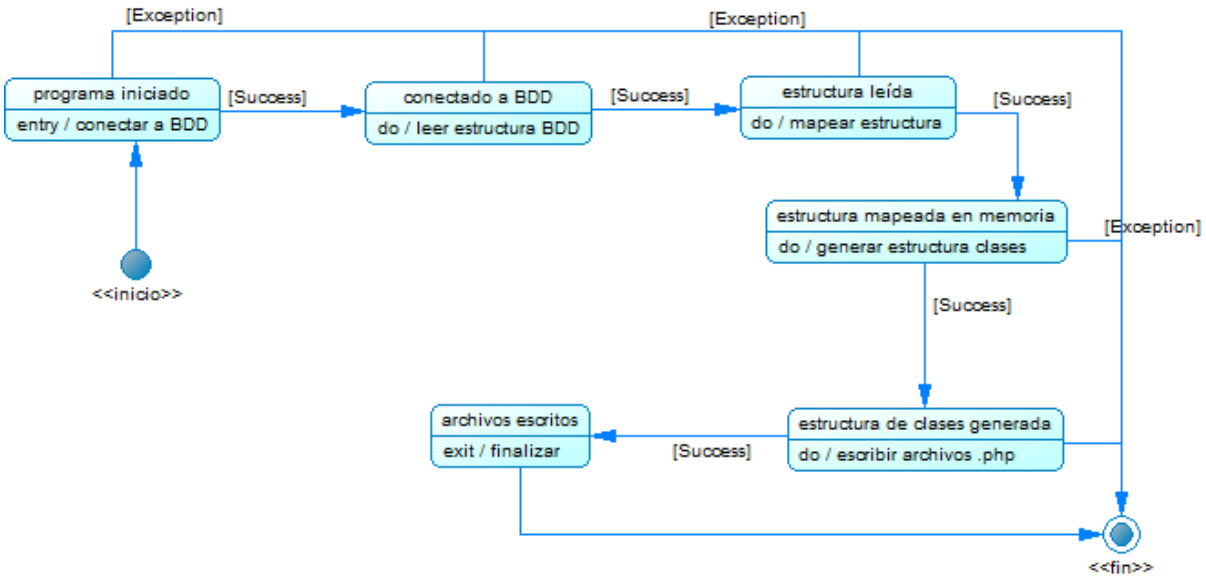

Ilustración 22 Diagrama de estados

Para detallar completamente el funcionamiento de la librería y la interacción de las clases entre sí se realizan los diagrama de secuencia, mostrados a continuación:

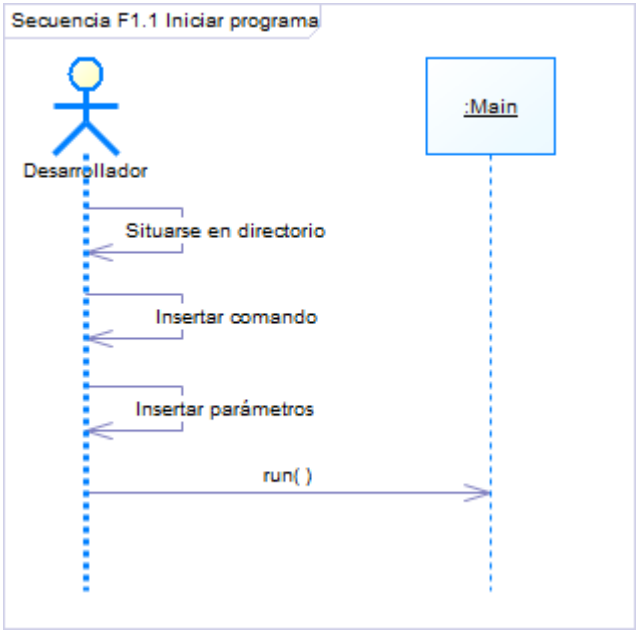

Ilustración 23 Diagrama de Secuencia: F1.1 Iniciar programa

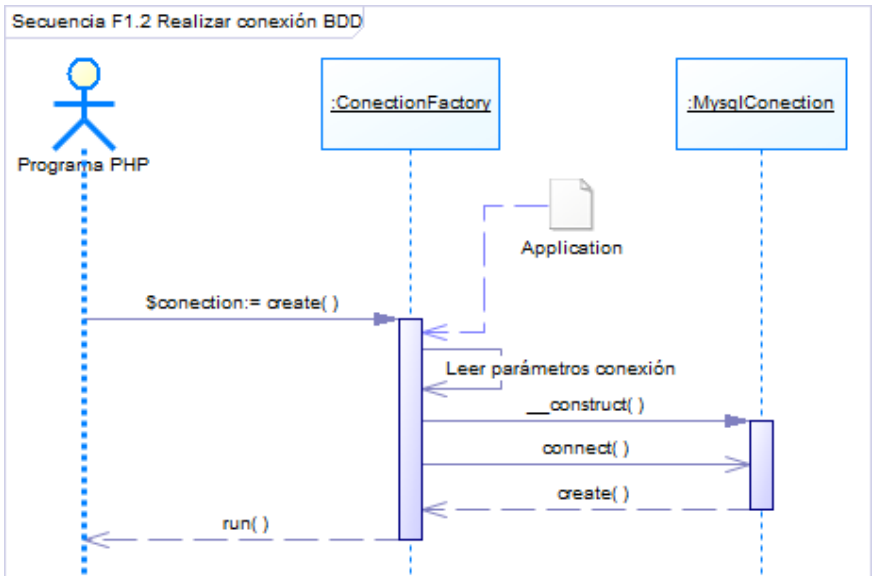

Ilustración 24 Diagrama de Secuencia: F1.2 Realizar conexión BDD

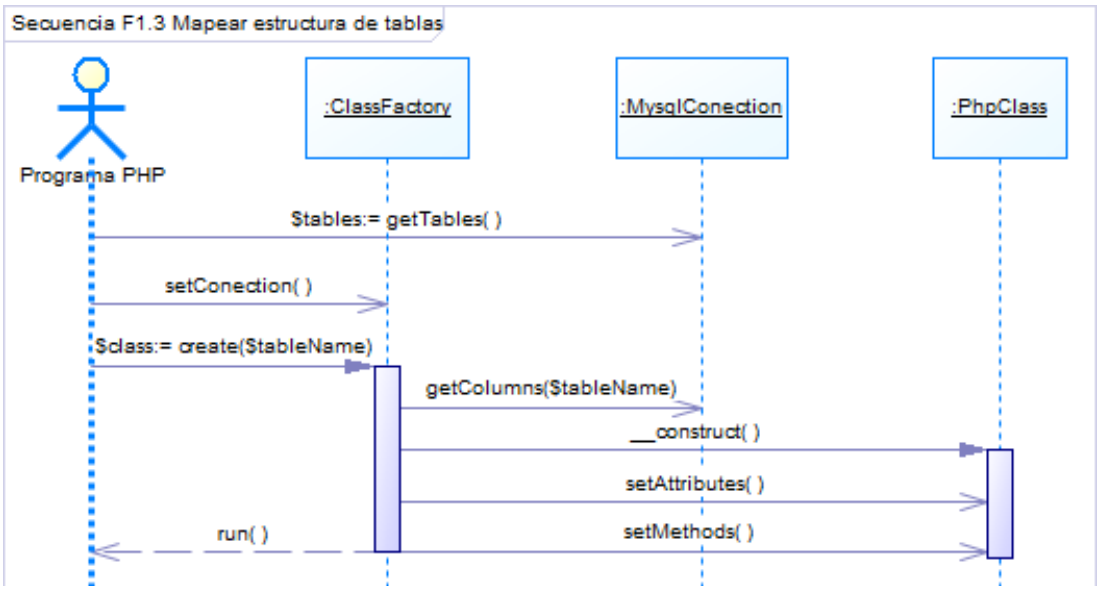

Ilustración 25 Diagrama de Secuencia: F1.3 Mapear estructura de tablas

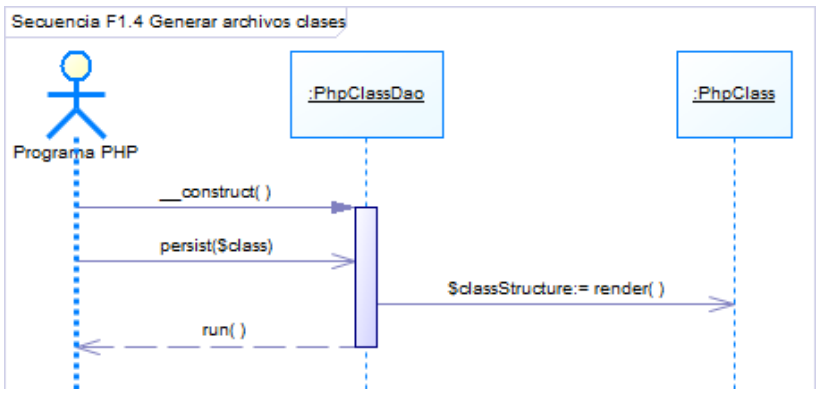

Ilustración 26 Diagrama de Secuencia: F1.4 Generar archivos clases

Finalmente para concluir el diseño de la librería es necesario especificar qué estándares de codificación se utilizarán durante la implementación. La siguiente ilustración muestra una clase de ejemplo que ilustra los estándares que se utilizarán para el desarrollo:

```php
<?php
/**
* Explanation of what the class handles.
*
* @author nameOfAuthor
*/
class ExampleClass extends ParentClass implements CommonInterface
{
	/**
	* Description of constant (if needed).
	* @var Data type of constant
	*/
	const CLASS_CONSTANT;

	/**
	* Description of attrbute (if needed).
	* @var Data type of attribute
	*/
	private $attribute;

	/**
	* Description of immutable attrbute (if needed).
	* @var Data type of immutable attribute
	*/
	private $_immutableAttribute;

	/**
	* Constructor description (if needed).
	*
	* @param DataType $param
	* @param DataType $optionalparam
	*/
	function __construct($param, $optionalParam = 'defaultValue')
	{
		$this->attribute = $param;
		$this->_immutableAttribute = $optionalparam;
	}

	/**
	* Method description (if needed).
	*
	* @param DataType $param
	* @param DataType $optionalparam
	* @return DataType Return value description
	*/
	public function complexNameMethod($param)
	{
		// Do stuff...
		// Onnline comment

		/*
		* Multiple line
		* comment
		*/
	}
...
```

Ilustración 27 Clase de ejemplo de estándares de codificación

4.3 Implementación

A continuación se muestra un extracto del producto generado durante la fase de implementación. La librería completa se encuentra en el adjunto digital.

```php
<?php namespace Generator\Connection;

use Monolog\Logger,
	Monolog\Handler\StreamHandler,
	Generator\AbstractFactory;

class ConnectionFactory implements AbstractFactory
{
	private static $log;

	private static function init()
	{
		self::$log = new Logger(get_called_class());
		self::$log->pushHandler(new StreamHandler(Logger::DEBUG));
	}

	public static function create($db = null)
	{
		self::init();
		try
		{
			$path = __DIR__ . '/../../config/application.php';

			if (!file_exists($path))
			{
				throw new \Exception("No config file found!");
			}
			$config = require($path);
		}
		catch (\Exception $e)
		{
			self::$log->addError($e->getMessage());
			die($e->getMessage());
		}

		$mysqlConnection = new MysqlConnection(
			$config['db']['hostname'],
			$config['db']['port'],
			$config['db']['username'],
			$config['db']['password'],
			$config['db']['schema'],
			$config['db']['name']
		);

		self::$log->addInfo('Attempting to connect to database');

		if ($mysqlConnection->connect())
		{
			self::$log->addInfo('Now connected to database');
			echo 'Now connected to database', PHP_EOL;
			return $mysqlConnection;
		}

		self::$log->addError('Error connecting to database');
		die("Error connecting to database\n");
		return null;
	}
}
```

Ilustración 28 Extracto de código: ConnectionFactory.php

El extracto mostrado más arriba muestra la clase que maneja la creación de objetos de conexión a la BDD. Es un fragmento interesante porque muestra cómo un paso intermedio controla toda la lógica de creación de un objeto complejo y que requiere varias validaciones, mientras que en la clase principal solamente se llama a un método creador que devuelve el objeto listo (`$connection = ConnectionFactory::create();`).

Es una muestra de los amplios beneficios de usar patrones de diseño en el desarrollo de software. La separación de responsabilidades ayuda a que el mantenimiento futuro se vuelva mucho más fácil e inclusive si se requiere extender la funcionalidad para otras BDD se lo puede realizar fácilmente.

4.4 Pruebas

Los casos de prueba y sus resultados se muestran en la tabla a continuación:

Caso de Uso	Precondición	Entradas	Resultado esperado	Cumple
F1.1	N/A	Comando Correcto	Librería inicia	**SI**
		Comando Incorrecto	Mensaje de ayuda. Se corta la ejecución	**SI**
F1.2	Archivo de configuración con parámetros de conexión a BDD	Parámetros de conexión correctos	Conexión se realiza con éxito. Mensaje de confirmación	**SI**
		Parámetros de conexión incorrectos	Conexión no se puede realizar. Mensaje de error	**SI**
F1.3[39]	Conexión a BDD establecida	Estructura de tablas de BDD	Mensaje de confirmación de estructura leída y mapeada	**SI**
F1.4	Estructura de BDD mapeada en memoria	Estructura de clase en memoria	Mensaje de confirmación de éxito	**SI**
		Estructura de clase en memoria. Directorio sin permiso de escritura	Mensaje de error de escritura	**SI**

Tabla 5 Casos de prueba de sistema

[39] Debido a que la BDD gestiona la integridad de datos, se asume que la estructura leída no posee ambigüedad ni fallos.

Las pruebas de la librería se pueden ver a mayor detalle en el adjunto digital. En la carpeta de pruebas se puede encontrar la ejecución de las pruebas en formato de video.

4.5 Postmortem

Una vez realizado el producto siguiendo la metodología PSP se han obtenido los siguientes datos (tabla 6) acerca del tiempo utilizado[40]:

	Minutos	Horas
Tiempo total estimado:	3065	51:05
Tiempo total real:	2498	41:38
Interrupción Total:	374	6:14
Interrupción máx.:	188	3:08
Interrupción min.:	0	0:00
Interrupción media:	15.58	0:16
Desviación Total:	567	9:27
Desviación máx.:	179	2:59
Desviación min.:	-24	-0:24
Desviación media:	23.63	0:24
Error Total:	22.70%	
Error máx.:	147.93%	
Error min.:	0.00%	
Error medio:	41.01%	

Tabla 6 Resumen de log de tiempo

Como se aprecia en la tabla anterior el tiempo total neto utilizado sobrepasa las 41 horas. Este tiempo cae dentro de la estimación inicial realizada que estaba en el rango de 36 a 44 horas de trabajo neto. Por otro lado el tamaño del producto llega a 47 hojas, sin tomar en cuenta el análisis posterior al producto, muy cerca de las 48 hojas estimadas inicialmente.

En cuanto al tamaño en líneas de código se estimó un rango de 1140 a 1260 LOC, el tamaño real de la librería en LOC se muestra en la tabla a continuación:

[40] Para el detalle del log de tiempos referirse al anexo 7.1

	LOC
Total:	956
Sin espacios en blanco:	789
Sin comentarios:	647

Tabla 7 Conteo de LOC

Finalmente, se presentan los siguientes gráficos para entender de mejor manera los resultados de tiempo obtenidos al aplicar PSP:

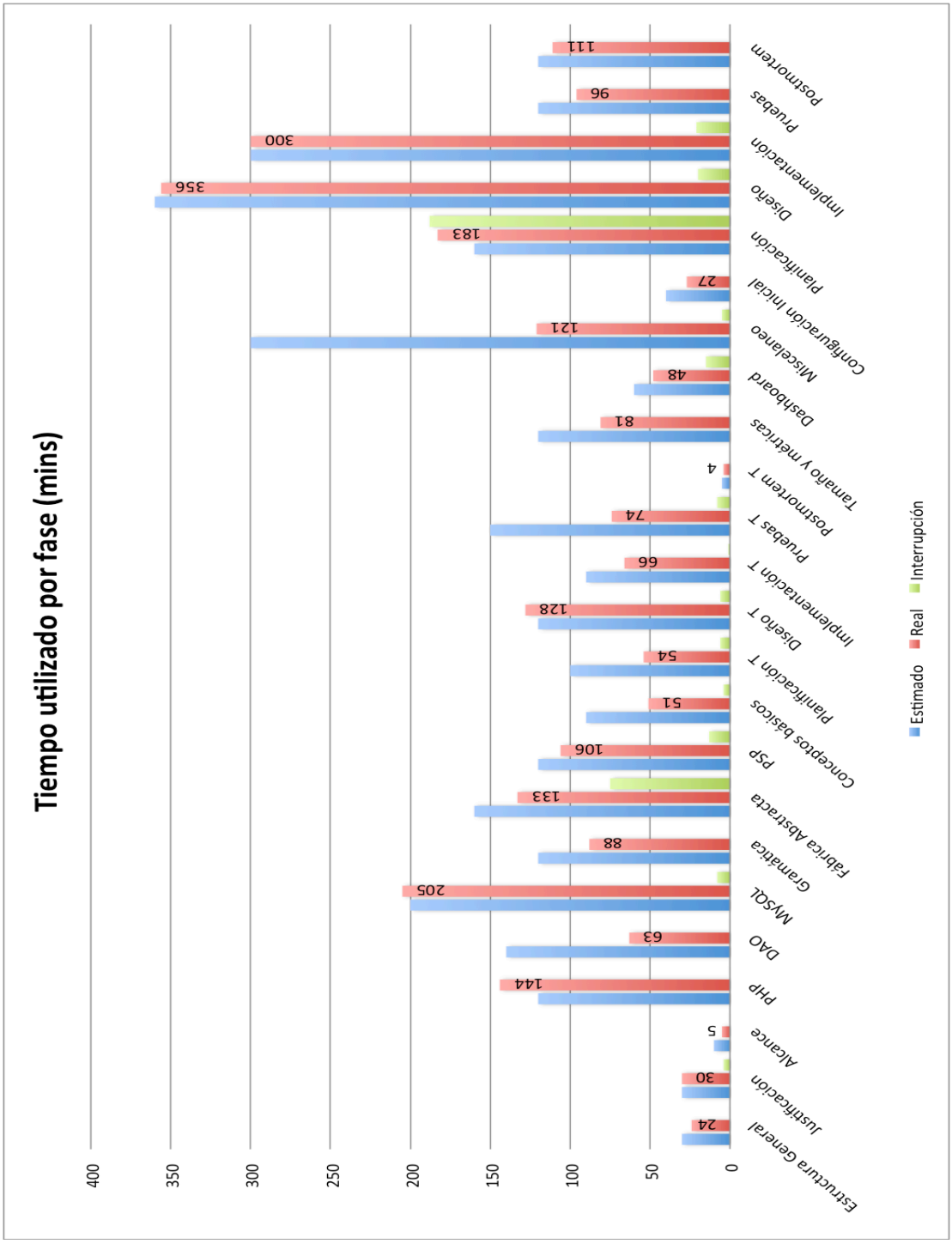

Ilustración 29 Tiempo utilizado por fase

En el gráfico anterior se muestra el tiempo utilizado por fase en minutos, tanto el tiempo real como el estimado y las interrupciones.

A continuación se muestra el progreso del producto en tiempo:

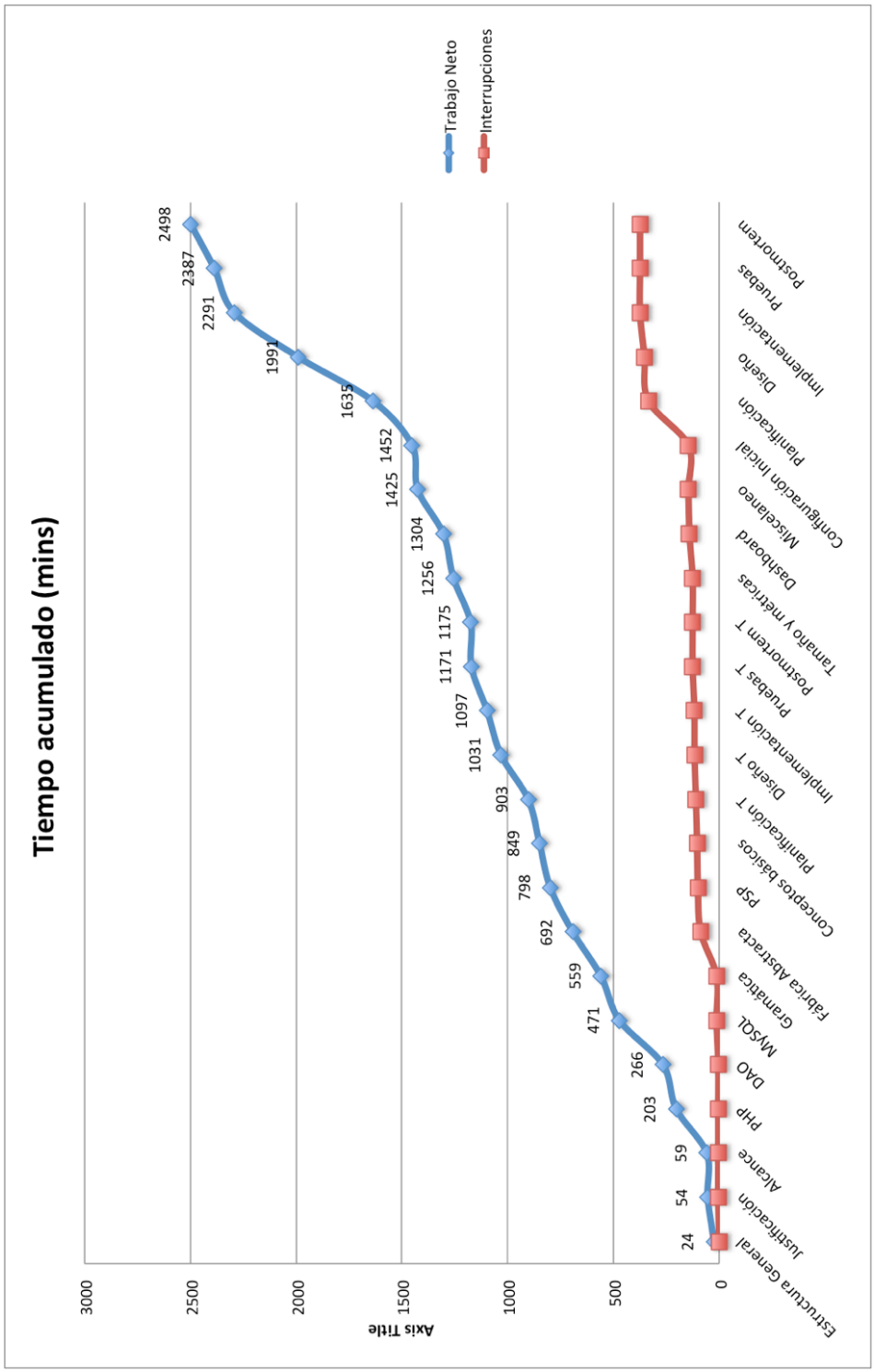

Ilustración 30 Gráfico tiempo acumulado por fase

Finalmente se muestra la proporción que representa cada fase con respecto al total de tiempo utilizado, en dos gráficos:

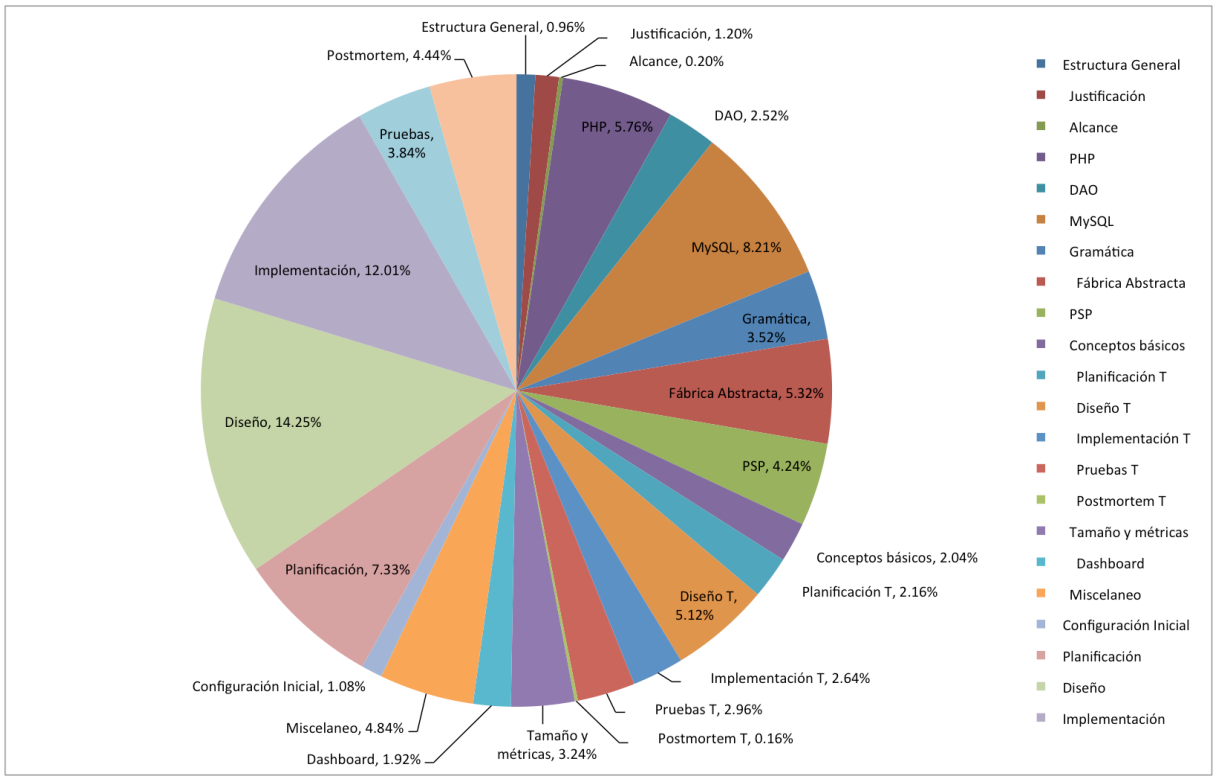

Ilustración 31 Gráfico proporción por fase

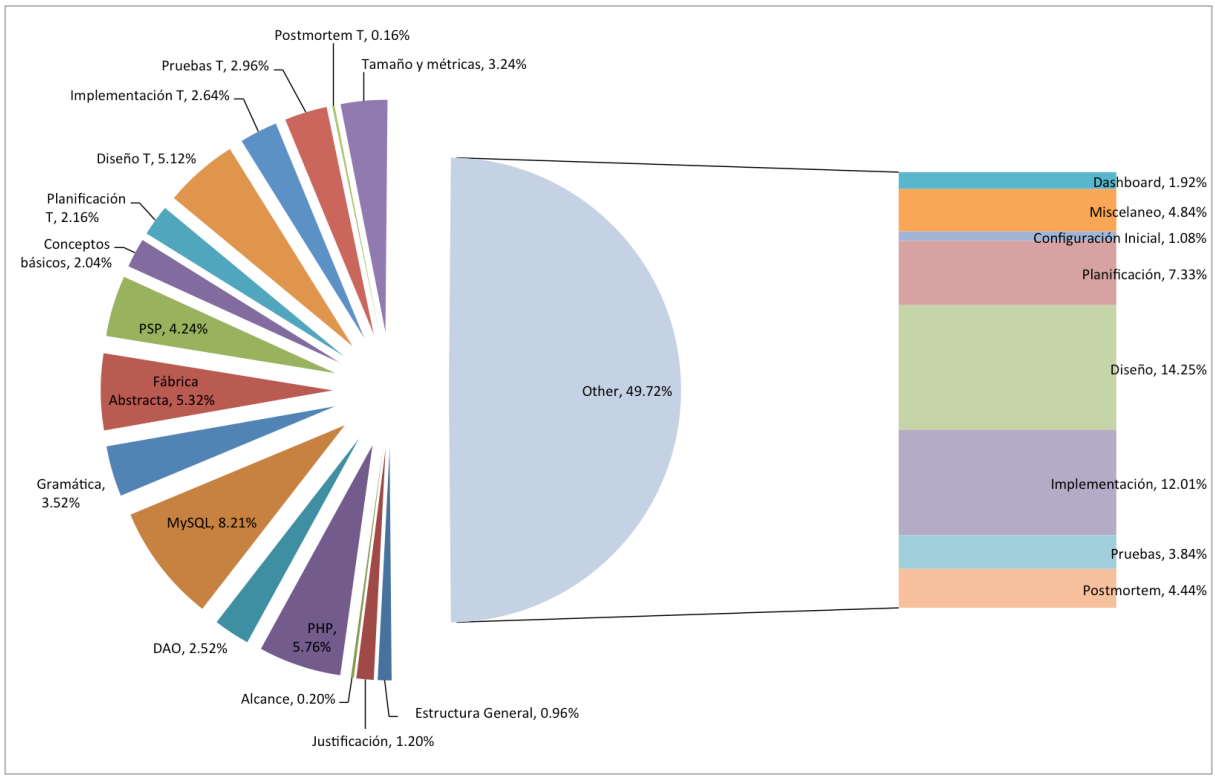

Ilustración 32 Gráfico proporción por fase separado

CAPÍTULO 5 CONCLUSIONES Y RECOMENDACIONES

En este capítulo se detallan las conclusiones del proyecto y las recomendaciones a tomar en cuenta para desarrollos en el futuro.

5.1 Conclusiones

Se construyó exitosamente la librería para la generación de clases PHP de forma automática leyendo directamente de una base de datos. Este producto se diseñó utilizando los criterios definidos por las buenas prácticas documentadas en patrones y empleando la metodología de desarrollo PSP.

Gracias a los de patrones de diseño utilizados, la aplicación es lo suficientemente flexible para que se pueda extender su funcionalidad fácilmente. Si se desea ampliar su uso para generar clases de otros lenguajes, es posible sin mayor esfuerzo ni reprogramación. Asimismo, si se necesita conectar a otras bases de datos que no sea MySQL se podría lograr sin necesidad de cambiar mayormente el programa sino extendiendo su funcionalidad e incluyendo el driver respectivo.

Además, ésto asegura que la librería sea totalmente orientada a objetos y que posee una base robusta que permite evitar errores comunes y vuelve su mantenimiento más sencillo. Esto se debe a que los conceptos de orientación a objetos se cumplen a cabalidad, sobretodo la separación de responsabilidades.

La automatización de procesos repetitivos y tediosos favorecen la estandarización del código y evitan errores humanos. Si la creación de clases entidad se basa en la librería que las genera sin intervención del programador, se asegura que éstas no posean errores de escritura ni se salgan de los estándares de codificación establecidos.

Por otro lado, al aplicar la metodología PSP se logró que el tiempo de desarrollo y pruebas sea pequeño en comparación con el tiempo total del proyecto. La carga más alta para una fase individual corresponde al diseño de la librería que llegó al 14.25% del tiempo total del trabajo, es decir, a 5 horas 56 minutos.

Un beneficio importante de la metodología es la reducción del tiempo de pruebas del producto mediante la aplicación de revisiones de código periódicas a lo largo de la

fase de implementación. De esta forma se llegó a una duración de la fase de pruebas correspondiente al 3.84% del tiempo total del proyecto.

De igual manera, la fase de implementación se ve notablemente reducida porque el diseño se lleva la mayor importancia dentro del ciclo de vida del software. Para este caso particular la implementación se extendió al 12.01% del tiempo total y equivale a 5 horas de desarrollo frente a una duración total del proyecto de 41 horas y 38 minutos.

Esto demuestra que la correcta aplicación de una metodología equilibra el tiempo que se utiliza para cada fase del ciclo de vida del producto y queda claro que la implementación, no es el paso que más trabajo conlleva dentro del ciclo de vida de desarrollo de software.

Para finalizar es importante mencionar que el soporte de la comunidad es un punto crucial para el proceso de maduración del producto, por lo tanto el código de la librería ha sido liberado y se lo puede encontrar en BitBucket[41]; se lo puede clonar o hacer fork de manera pública.

5.2 Recomendaciones

Es recomendable utilizar la librería para la generación de modelos en implementaciones del patrón Modelo-Vista-Controlador (MVC). Para dar un ejemplo práctico, se lo puede utilizar para complementar el ORM del framework Kohana (v3.3.*) que actualmente no posee una herramienta para crear modelos, sino que se los debe programar manualmente.

Por otro lado para una nueva versión de la librería producida, sería aconsejable tomar en cuenta dos puntos de ampliación de la funcionalidad.

El primero es dar soporte a un mayor número de bases de datos. Para lograrlo, se debe incluir un driver más robusto que el utilizado actualmente, una buena opción es el driver PDO, porque además de estandarizar las interfaces de acceso a la BDD, sea cual sea el motor, permite conectarse con casi cualquier base de datos comercial actual.

[41] https://bitbucket.org/jltorresm/class-generator/

El segundo punto de ampliación, es extender el alcance de la librería para que funcione como ORM. Esto quiere decir, que ya no solo generaría modelos sino que permitiría utilizarla como una herramienta de mapeo objeto-relacional para leer y escribir datos de una manera orientada a objetos.

Finalmente, para la Facultad de Ingeniería de la PUCE se recomienda revisar los contenidos de la carrera de Ingeniería de Sistemas, los cuales deberían reflejar las nuevas tendencias de tecnología. Asimismo fomentar una educación orientada a la investigación y autoaprendizaje porque la universidad no puede darlo todo, sino que es trabajo del estudiante seguir en un proceso de mejora continua una vez acabados los estudios.

Es crítico, además, que la educación se base en brindar nuevos conceptos y no en enseñar herramientas específicas, para no limitar el campo de trabajo del estudiante. De esta manera se logra formar profesionales que sean capaces de aplicar los conceptos a cualquier herramienta y dentro de equipos de trabajo heterogéneos, típicos de el campo laboral global.

BIBLIOGRAFÍA

De La Cruz, Fabián. *Patrones.* Clase Magistral, Quito: Pontificia Universidad Católica del Ecuador, 2013.

Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

Grune Dick, van Reeuwijk Kees, Bal Henri E., Jacobs Ceriel, Langendoen Koen. *Modern Compiler Design.* 2da. New York: Springer, 2012.

Humphrey, Watts. *The Personal Software ProcessSM PSPSM.* Madrid: Pearson Education S.A., 2001.

ISEG. *Data Access Object.* Vers. 1. Information & Software Engineering Group. 4 de Octubre de 2013. http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/dao.html (accessed 16 de Marzo de 2014).

Kulkarni, Virendra. «Understanding Factory Method and Abstract Factory Patterns.» *Code Project.* 24 de Abril de 2013. http://www.codeproject.com/Articles/35789/Understanding-Factory-Method-and-Abstract-Factory (accessed 27 de Marzo de 2014).

Marco, Bartolomé Sintes. *Qué es PHP.* 17 de Septiembre de 2013. http://www.mclibre.org/consultar/php/lecciones/php_quees.html (accessed 13 de Enero de 2014).

Mehdi Achour, Friedhelm Betz, Antony Dovgal, Nuno Lopes, Hannes Magnusson, Georg Richter, Damien Seguy y Jakub Vrana. *History of PHP.* 20 de 11 de 2013. http://www.php.net/manual/en/history.php.php (accessed 16 de 03 de 2014).

Michael, Kofler. *The Definitive Guide to MySQL 5.* 3era. Translated by Kramer David. New York: Springer-Verlag Inc., 2005.

Pomeroy-Huff, Marsha. «The Personal Software ProcessSM (PSPSM) Body of Knowledge.» *Software Engeneering Institute.* Carnegie Mellon University. 01 de Agosto de 2009. http://www.sei.cmu.edu/reports/09sr018.pdf (accessed 13 de Junio de 2013).

Rumbaugh, James, Ivar Jacobson y Grady Booch. *El Lenguaje Unificado de Modelado Manual de Referencia.* 2da Edición. Madrid: Pearson Educación S.A, 2007.

Schwartz Baron, Zaitsev Peter, Tkachenko Vadim. *High Performance MySQL.* 3era. Sebastopol : O'Reilly Media, Inc. , 2012.

Sommerville, Ian. *Ingeniería Del Software.* 7ma. Madrid: Pearson Educación S.A., 2005.

Stelting Stephen, Maasen Olav. *Patrones de Diseño aplicados a JAVA.* Translated by Campos Martínez Manuel. Madrid: Pearson Education, 2003.

Sweat, Jason. *Php|Architect's Guide to PHP Design Patterns. A Practical Approach to Design Patterns for the PHP 4 and PHP 5 Developer.* Toronto: Marco Tabini & Associates, Inc., 2005.

Williams Michael, McGinn Tom, Heimer Matt. *Java SE 7 Programming.* 2da. Edited by Richard Wallis. Vol. 1. California: Oracle University, 2012.

ANEXOS

Detalle de Log de Tiempo

Jerarquía	Tiempo Estimado (mins)	Tiempo Real (mins)	Tiempo Real Acum.	Interrupción (mins)	Interrupción Acum.	Desviación	Error	% Avance
Estructura General	30	24	24	0	0	6	25.00%	0.96%
Justificación	30	30	54	4	4	0	0.00%	1.20%
Alcance	10	5	59	0	4	5	100.00 %	0.20%
PHP	120	144	203	0	4	-24	16.67%	5.76%
DAO	140	63	266	0	4	77	122.22 %	2.52%
MySQL	200	205	471	8	12	-5	2.44%	8.21%
Gramática	120	88	559	0	12	32	36.36%	3.52%
Fábrica Abstracta	160	133	692	75	87	27	20.30%	5.32%
PSP	120	106	798	13	100	14	13.21%	4.24%
Conceptos básicos	90	51	849	4	104	39	76.47%	2.04%
Planificación T	100	54	903	6	110	46	85.19%	2.16%
Diseño T	120	128	1031	6	116	-8	6.25%	5.12%
Implementación T	90	66	1097	1	117	24	36.36%	2.64%
Pruebas T	150	74	1171	8	125	76	102.70 %	2.96%
Postmortem T	5	4	1175	0	125	1	25.00%	0.16%
Tamaño y métricas	120	81	1256	0	125	39	48.15%	3.24%
Dashboard	60	48	1304	15	140	12	25.00%	1.92%
Miscelaneo	300	121	1425	5	145	179	147.93 %	4.84%
Configuración Inicial	40	27	1452	0	145	13	48.15%	1.08%
Planificación	160	183	1635	188	333	-23	12.57%	7.33%
Diseño	360	356	1991	20	353	4	1.12%	14.25%
Implementación	300	300	2291	21	374	0	0.00%	12.01%
Pruebas	120	96	2387	0	374	24	25.00%	3.84%
Postmortem	120	111	2498	0	374	9	8.11%	4.44%
TOTAL	3065	2498		374		567	41.01%	100.00%

Tabla 8 Log de tiempo a detalle