Proyecto de software

Salsamendi Jeremías

Índice

1. Introducción

En el presente trabajo se desarrollará el proceso de construcción de un software de administración web típico en base a la producción de uno en particular. Se explicará cómo se llevaron a cabo cada una de las partes del sistema, primero desde una visión lógica y conceptual, seguida por una práctica implementada en el lenguaje de programación Python. El sistema cuenta con un apartado de administración privado y uno de acceso a la información, público. Su función principal será la administración de aspectos relativos a la organización social de la ciudad de La Plata, provincia de Buenos Aires ante eventos inesperados de siniestros pluviales.

Asimismo, abordaremos todas las herramientas utilizadas en la implementación del sistema. Se hará especial hincapié en el beneficio que aportan y cuánto funcionamiento nuclean. Cada una de ellas resuelve un problema específico dentro del contexto del presente trabajo por lo que son no solo óptimas sino también necesarias.

2. Palabras clave

Aplicación web, sistema informático de administración, diseño de sistema, Flask, Vue

3. Desarollo

Para explicar de manera ordenada la construcción del sistema se detalla la estructura general del mismo. En base a ello, se podrán explicar otras funcionalidades satélite que se apoyan sobre dicha estructura.

2.1. Arquitectura Cliente-Servidor

El sistema al ser manipulado por los usuarios mediante el navegador web, debe tener una capa de abstracción subyacente que determine cómo funciona a grandes rasgos la interacción de las computadoras involucradas en el uso del mismo, para que haya consistencia en su funcionamiento. La arquitectura cliente-servidor es la utilizada en internet y por consiguiente la más adecuada para el problema a tratar. La misma define a una computadora (o también pueden ser múltiples, pero por simplicidad en este caso se la considera única) como proveedora de servicios y a otras que se comunican con ella como solicitantes de servicios. Estos dos roles son respectivamente los de cliente y servidor.

2.2. Arquitectura del servidor: Modelo-Vista-Controlador (MVC)

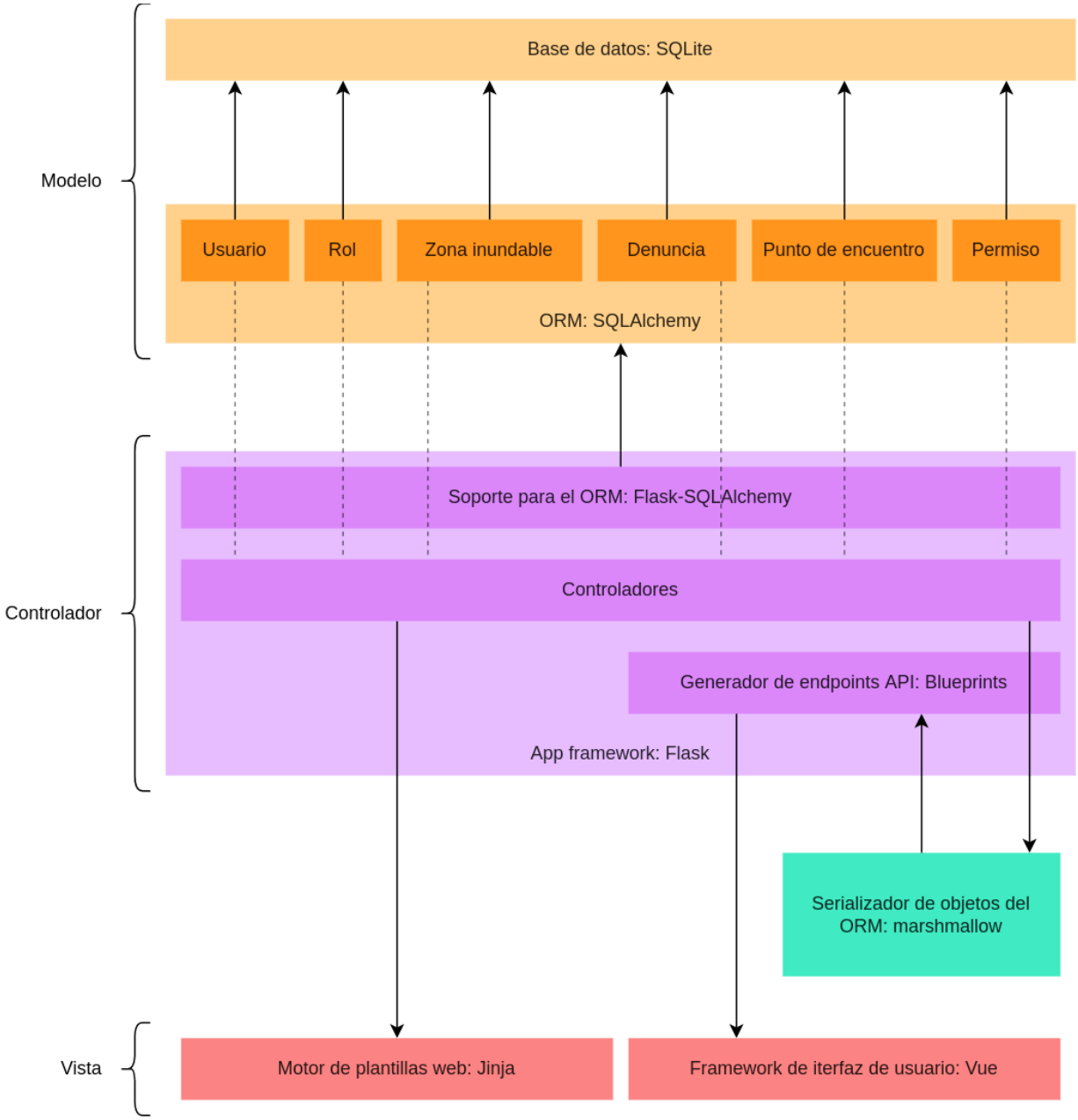
El funcionamiento del sistema de administración *web* definido sigue una arquitectura Modelo-Vista-Controlador (MVC). La misma tiene sus bases en la programación orientada a objetos y determina que el sistema se compone de tres elementos: el modelo, las vistas, y los controladores. El primero de ellos define la estructura de los datos persistentes a almacenar, mientras que los últimos definen la lógica de funcionamiento del sistema y las operaciones posibles a realizar. Por último las vistas permiten proporcionar al usuario acceso a las funcionalidades del software. La imágen 1 presenta el esquema general a analizar.

2.2.1. Estructura del sistema de archivos

Aunque la estructura a nivel de sistema de archivos sea independiente del proyecto y de la arquitectura MVC, es menester aclarar cómo está constituida para poder seguir adelante con ésta y las restantes explicaciones. En términos pragmáticos la organización del sistema de archivos articula todos los módulos del sistema. Es relevante hacerlo para poder ordenar el proyecto en el marco del sistema de archivos y de esta manera mantenerlo flexible, reusable y extensible. Para llevar adelante esta tarea se creó una carpeta de la aplicación llamada “app” y dentro de ella se generaron: el *script*[1] Python que ejecuta la aplicación principal (que brinda el servidor en escucha); un *script* que inicializa la base de datos y las carpetas Models, Schemas, Resources, Templates, Helpers, Static y Forms. Cada una de estas carpetas cumple una función específica y todas contienen archivos Python relacionados a dicha funcionalidad: en Models se definieron los modelos; en Schemas se definieron los esquemas de paginación; en la carpeta Resources se ubicaron los controladores por modelo; en Templates se definieron las plantillas; en Helpers se escribieron *scripts* de traducción de objetos a formato JSON (esto luego será explicado en la sección API REST); en Static se colocaron archivos de propósito general que son solicitados por los usuarios y deben ser enviados en las solicitudes

[1] En informática se considera a los scripts como archivos de texto con comandos que constituyen programas relativamente simples.

HTTP; y por último en la carpeta Forms se definieron los formularios del sistema que articulan la información ingresada en un formulario web con lo que luego se manipula en los controladores.



Imágen 1: vista general de la estructura del sistema

2.2.2. Modelo

El diseño del modelo tiene dos aspectos fundamentales: el apartado de la base de datos y la abstracción de dichos elementos al modelo orientado a objetos. El primero define de manera detallada los elementos persistentes del sistema y su relación con los aspectos físicos de su almacenamiento, mientras que el segundo facilita el uso de dichos elementos en etapas posteriores de la construcción.

2.2.2.1. Base de datos

La forma clásica y más ampliamente utilizada de llevar a cabo el diseño de la base de datos es siguiendo el modelo entidad-relación, mediante el cual se definen los elementos estructurales persistentes del sistema como tablas y relaciones entre ellas. Luego, mediante un sistema de gestión de bases de datos (SGBD) se realizan las operaciones (consultas, inserciones, modificaciones y eliminaciones) sobre dichas tablas. Este apartado del modelo se implementó mediante SQLite, un sistema de gestión de base de datos relacional que permite definir tanto la estructura, como realizar la operaciones sobre la base de datos.

2.2.2.2. Mapeo objeto-relacional (ORM)

Para incorporar y luego poder usar el SGBD se utiliza ORM (Object-Relational mapping), una técnica para traducir el diseño relacional al modelo orientado a objetos y poder utilizarlo desde el lenguaje Python. La técnica se lleva adelante mediante SQLAlchemy, una herramienta que permite establecer una relación entre los elementos del modelo orientado a objetos - que se manipulan en el lenguaje de programación - y el relacional operado por el SGBD. De esta manera, en Python se definen clases por cada elemento de la base de datos y se establecen relaciones entre cada una y la tabla del modelo relacional que le corresponde, sus métodos son operaciones solicitadas a SQLite. Tanto la relación clase-tabla como la traducción de los métodos en consultas SQL (solicitudes que se realizan a los SGBD para la manipulación de las bases de datos) son efectuadas por el ORM, SQLAlchemy.

Retomando el proyecto de siniestros pluviales, para la implementación de los modelos del sistema se creó una nueva carpeta en el proyecto, y en ella se definieron módulos Python por cada uno de los elementos de la base de datos. Se crearon las clases User, Role, Flood zone, Meeting point, Permit, Complaint, Evacuation route, Statistics, Category, Config, Color y Palette; cada una en su respectivo módulo. Al definirlas se debió establecer una relación con su correspondiente elemento de la base de datos. Haciendo uso del ORM SQLAlchemy y del paradigma orientado a objetos, se definieron los métodos de clase que operan sobre el conjunto total de instancias de cada elemento e interactúan con el SGBD, y los métodos de instancia que operan sobre los ejemplares de cada grupo de elementos.

2.2.3. Controlador

Los controladores articulan la información persistente del sistema con el usuario e introducen las funcionalidades generales y estructurales del sistema. Son los elementos que conectan y relacionan los diferentes componentes, conformando al sistema. En términos pragmáticos son conjuntos de *scripts* en el servidor que responden a las distintas solicitudes HTTP, recuperando objetos (de repositorios o bases de datos) del modelo y operando con ellos para otorgar una respuesta que el usuario recibe mediante las vistas en forma de páginas *web*.

2.2.3.1. Framework de aplicaciones web

El funcionamiento de los controladores conlleva una coherencia y compatibilidad entre las diferentes partes que componen al sistema, lo que permite introducir el concepto de *framework web*. En principio un *framework* es una aplicación semi-completa reutilizable, que para ser completa solo necesita que se le definan ciertos aspectos particulares (pensados por su diseñador). Su objetivo es proveer una arquitectura reutilizable para toda una familia de aplicaciones. Dichos aspectos a definir

son llamados ganchos (*hooks*) y es mediante éstos que el usuario del *framework* (note el lector la diferencia con el usuario final del sistema) puede articular su desarrollo y caracterizar a la aplicación que está realizando dentro del universo de las abarcadas por el *framework*. Por otro lado, que sea *web* implica que funciona en el contexto de una red de comunicación remota entre computadoras. Aquí se retoma la arquitectura cliente-servidor y cobra importancia su uso.

El *framework* utilizado para la realización de este sistema es Flask. Es minimalista y extensible, y permite ampliar su funcionamiento siempre que sea necesario.

Para la implementación de los controladores se creó una nueva carpeta con módulos Python que tuvieran los nombres de los elementos de la base de datos. Se los agrupó de esta forma a modo de mantenerlos ordenados y poder accederlos fácilmente. Estos *scripts* se completaron con métodos que definen el comportamiento del sistema, tanto ante las operaciones que se puedan realizar sobre cada modelo como ante operaciones que tengan que ser realizadas relacionando múltiples partes del sistema no conexas. Entre dichas operaciones se encuentran las operaciones básicas: alta, modificación, baja y búsqueda mediante diferentes criterios.

2.2.3.2. API REST

Para el presente proyecto resultó útil incorporar una API (*Application Programmable Interface*, Interfaz de Programación de Aplicaciones) para poder proporcionar la información desde el sistema a otros, y de esta forma hacer un apartado público del sistema completamente independiente del sistema base descripto hasta el momento. API es una tecnología que utiliza un conjunto de protocolos y estándares para definir una forma de comunicación entre sistemas distribuidos y/o descentralizados para lograr interoperabilidad. En este caso en particular la interoperabilidad se ve reflejada, en principio, en la utilización de los recursos que proporciona Flask por el *framework* de interfaz de usuario Vue en la aplicación pública. El método de comunicación es en formato JSON (JavaScript Object Notation: Notación de objetos JavaScript) el cual utiliza texto plano, es de fácil manipulación y es ampliamente utilizado por lo que tiene mucho soporte[2].

La API, a su vez, es de tipo REST (*Representational State Transfer*, Transferencia de estado representacional) lo que indica que se apoya completamente sobre el estándar HTTP y funciona en la *web*, por lo que su conexión es sin estado. Por esto mismo, también se debe tener en cuenta el concepto de cookies que son de mucha utilidad para poder generar en los nodos un estado en la comunicación aunque realmente en la conexión HTTP no exista. Los elementos que se soliciten tienen su propia URL y pueden ser fácilmente cacheados, copiados y guardados.

La API REST se definió en una nueva carpeta dentro de Resources, donde se especificaron mediante Blueprints, una funcionalidad propia de Flask, los *endpoints* deseados del sistema. Un *endpoint* es un punto (en este caso ubicación mediante URL) donde se depositan o ubican los elementos a solicitar de la API. De esta manera se generaron las respectivas URL y mediante Marshmallow (un *micro-framework* que permite traducir objetos a formato JSON) los elementos son traducidos para que al solicitar el recurso HTTP se obtengan datos en formato JSON.

2.2.4. Vista

Para entender las vistas se necesita socavar en el funcionamiento de una petición HTTP y la relación entre ésta, los controladores y la forma de mostrar la información.

[2] Un elemento, en términos informáticos, tiene soporte cuando es ampliamente usado y la comunidad provee múltiples herramientas para su uso y manipulación.

Al hacer una solicitud a una URL, la misma es en principio atendida por su correspondiente controlador del lado del servidor. Es mediante el controlador que el servidor determina qué acciones llevar a cabo y, en ciertas circunstancias, qué mostrar al usuario luego de ésto. Como se acaba de mencionar, no siempre los controladores terminan su ejecución enviando información a ser mostrada sino que también pueden resolverse haciendo una llamada a otro controlador que sí envía información visual al navegador cliente.

Teniendo en cuenta lo anterior, el muestreo de información se puede llevar a cabo de dos formas, de manera estática, es decir, enviando la página HTML (HyperText Markup Language: Lenguaje de marcado de hipertexto) a ser impresas por el navegador por cada llamada a un controlador; o de manera dinámica teniendo un mismo HTML con un *script* JavaScript que responda bajo demanda a las acciones del usuario y haga las peticiones a la API REST sin hacer una nueva reimpresión de todo un HTML y modificando el DOM (*Document Object Model*, Modelo de objetos del documento) actual.

2.2.4.1. Interfaz de usuario estática

Las plantillas correspondientes a la interfaz de usuario estática se ubicaron en la carpeta Templates. Para poder definirlas de manera genérica, independientemente de los modelos, se las estructuró de forma que el contenido de las mismas dependiera de los parámetros que se les envíe abstrayendo datos específicos. Además se hace uso del motor de templates de Flask, Jinja versión 2, que permite añadir ciertas porciones de programa imperativo en la plantilla, como estructuras de control, conservando la generalidad y abstrayendo lo particular del HTML mostrado por el navegador en un momento dado. Posteriormente el motor de templates resuelve esas porciones de programa para generar el HTML que se envía por la red. Cabe aclarar que el motor de templates funciona del lado del servidor, es decir, genera los documentos HTML y luego los envía mediante la red al cliente, y aquí mismo se encuentra la diferencia con la alternativa estática, donde el HTML es manipulado por el lado cliente.

Para llevar esto adelante se estructuró el sistema de archivos de la siguiente forma, se definió dentro de la carpeta Templates una carpeta con nombre Generic donde se ubicaron las plantillas con funcionamiento genérico del sistema y por otro lado se definieron las carpetas de cada uno de los elementos del modelo por si se necesitaba especificar funcionamiento de plantillas particulares en algún caso. Luego en Generic se definieron dos tipos de plantillas y para ello se crearon las carpetas Elements (donde se alojan los elementos que se ven en las páginas) y Pages (las propias páginas constituidas por elementos). Una vez estructurado de esta forma se definieron los HTML como correspondía.

2.2.4.2. Interfaz de usuario dinámica: Framework de interfaz de usuario

Para la interfaz de usuario dinámica se hace uso de un *framework* de interfaces de usuario, Vue, el cual permite generar una aplicación de manejo de vistas del sistema. Este enfoque de solución al problema de muestreo de información al usuario permite mejorar el tiempo de respuesta, mantener actualizada la información que se muestra con los datos del servidor y enriquecer la experiencia del usuario.

Para la definición de rutas URL mediante las cuales acceder a las diferentes páginas se utilizó el *plugin*[3] Vue Router. Con él se pueden establecer fácilmente las direcciones a ser accedidas y

[3] Un plugin en el contexto de la informática es un complemento de un programa que extiende sus funcionalidades originales.

además las incorpora a su funcionamiento de modo que la interfaz visual sigue funcionando mediante JavaScript con una única petición de un HTML base.

Para el uso de este *framework* se creó una nueva carpeta al mismo nivel (en cuanto a sistema de archivos) que App llamada Web, para que contenga todo el contenido necesario para que pueda funcionar. Tanto esta creación como la instalación dentro de la carpeta, es realizada de manera automática mediante el gestor de paquetes JavaScript, llamado npm. Dentro de la carpeta Web en el script *route.js* se ubicaron las rutas del sistema público, en src/components se colocaron los elementos que componen las páginas y en src/pages las páginas que los utilizan.

2.3. Aplicación web progresiva (PWA)

Las aplicaciones web progresivas incorporan a una aplicación web clásica o tradicional la posibilidad de parecer nativas. Esto implica que puedan notificar mediante el sistema operativo al usuario de la información entrante, mantener información almacenada en el dispositivo de manera *offline* y acceder a ella por fuera de un navegador, incluso sin ingresar la URL.

Una PWA se compone de un *service worker*, *script* que funciona de *middleware*[4] entre las interacciones web de la aplicación del lado cliente y la red; y un archivo manifiesto donde se almacena la metainformación de la aplicación, es decir, datos acerca del sistema, pero que no son utilizados por éste. El *service worker* tiene un ciclo de funcionamiento: registro, instalación y activación. En el registro se incorpora el *service worker* para que comience su ejecución, en la instalación se inicializa y puede comenzar a almacenar en la caché, y finalmente se activa para luego quedar a la espera de solicitudes tanto entrantes como salientes.

En el apartado público, la PWA fue implementada de forma relativamente simple dado que se hizo uso del *plugin* pwa del *framework* Vue, que tiene incorporadas las políticas de uso de la caché e internet y muchas otras funcionalidades básicas que permiten un uso eficiente del *service worker* y de los recursos necesarios para la aplicación. Luego de instalar el *plugin*, se agregó un nuevo archivo en la carpeta web con el nombre vue.config.js el cual es reconocido automáticamente por Vue y dentro de él se definieron los datos del archivo manifiesto. Por otra parte se creó un nuevo *script* en la carpeta web/src con nombre service-worker.js donde se definieron qué políticas de uso de red y almacenamiento en caché se utilizan para cada tipo de solicitud.

2.4. Pruebas de unidad por modelo (Model unit testing)

La etapa de pruebas de unidad consiste en definir un conjunto de pruebas que permitan determinar si un módulo funciona de manera correcta y de este modo encontrar problemas en el sistema de una manera eficaz y relativamente rápida. Para este caso particular se analiza cómo hacer las pruebas de unidad sobre modelos de la base de datos del sistema. Para ello se prueban las clases del ORM en términos de sus métodos.

Para implementarlo se creó una nueva carpeta dentro de app que se llama tests y también se creó un nuevo archivo con nombre __init__.py. Luego, dentro de éste último, se hizo uso de la librería integrada en Python unittest que sigue el modelo de pruebas de unidad orientado a objetos en TDD[5], el cual permite definir clases que prueban módulos del sistema (en este caso los modelos).

[4] Un *middleware* es un *software* que actúa de intermediario entre otros (generalmente dos) componentes de *software* para hacer posible su comunicación o interacción.

[5] TDD hace referencia a *Test Driven Development*, una metodología de construcción de software en el contexto de la orientación a objetos en la que se realizan los objetos que prueban y los objetos a ser probados.

Cada clase de prueba posee un estado inicial que le permite garantizar un entorno de ejecución a los métodos de prueba de la unidad. Mediante esta librería se definió la clase base BaseTestModel que inicializa la aplicación y la base de datos. Luego deben ser creadas las tablas de la base de datos y para ello se definió un método abstracto en BaseTestModel que es invocado en su inicialización. Análogamente para la eliminación de las tablas se definió otro método abstracto. El hecho de que la creación de tablas suceda según lo necesite la unidad a probar, permitió un mayor control sobre la información involucrada en cada prueba y permitió evidenciar más eficazmente los posibles errores presentes en el sistema. Posteriormente se creó un archivo Python por cada modelo a ser probado y dentro se definen subclases de BaseTestModel que son las que prueban efectivamente las clases del ORM y deben definir los métodos abstractos de creación y eliminación de las tablas de la base de datos, y los que realizan las pruebas. Finalmente para comprobar que el modelo funcionaba correctamente se ejecutó el *script* __init__.py en entorno de pruebas de Flask y se obtuvo el resultado.

4. Consideraciones Finales

En síntesis, el proceso de creación de un sistema informático de administración integra múltiples disciplinas de la informática, que, para tener mayor control y seguridad en lo que se está desarrollando, deben ser estudiadas en profundidad y de esta forma poder utilizar las herramientas disponibles de una manera más eficaz, adecuada y eficiente. Por otro lado también es de gran utilidad poder esquematizar y bosquejar estructuralmente los sistemas a modo de definir un camino a seguir y poder establecer una base sólida en el proceso de construcción.

5. Anexo: impacto ético sobre del trabajo

Para la construcción del sistema se utilizaron componentes de software gratuitos con licencias libres (el software puede ser modificado, estudiado y utilizado libremente), en particular MIT y GPL. Esto garantiza un mayor acceso a las metodologías y conceptos de la informática, y a la implementación de ellas en las herramientas de producción.

Además se tuvo en cuenta la utilización de recursos de accesibilidad web en el apartado visual estático (ya que es donde más control se puede tener sobre la información y su organización), para ampliar el espectro de usuarios del sistema de administración.

6. Referencias

- Kurose James F., Ross Keith W. 2017. *Redes de computadoras: Un enfoque descendente*, 7ma ed. Madrid, España. Pearson.
- Pressman Roger S. 2010. *Ingeniería del software: Un enfoque práctico*, 7ma ed. México. McGraw-Hill.
- Sommerville Ian. 2011. *Ingeniería de software*, 9na ed. México. Addison Wesley.
- Ducasse Ducasse, Pollet Damien. 2017. Learning Object-Oriented Programming, Design and TDD with Pharo. Recuperado del día 10 de marzo de 2022, de https://files.pharo.org/books-pdfs/learning-oop/2017-09-29-LearningOOP.pdf.
- Fayad Mohamed, Schmidt Douglas C. 1997. Object-Oriented Application Frameworks. Recuperado el día 15 de marzo de 2022, de https://dl.acm.org/doi/pdf/10.1145/262793.262798.