Universidad de las Ciencias Informáticas
Facultad 3

"Módulo Salvar-Cargar del simulador para la Industria Química desde el rol de diseñador de sistema"

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Enaide Corrales Fernández

Tutor: Ing. Arnaldo Gandol Álvarez

Co-tutor: Dr. Francisco Andrés Cano Alonso

Junio 2007

DECLARACIÓN DE AUTORÍA

Declaro que soy el único autor de este trabajo y autorizo a la Facultad 3 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmo la presente a los ____ días del mes de _____ del año _____.

Enaide Corrales Fernández	Ing. Arnaldo Gandol Álvarez	Dr. Francisco Andrés Cano Alonso
_____	_____	_____
Firma del Autor	Firma del Tutor	Firma del Cotutor

DATOS DE CONTACTO

“Ing. Arnaldo Gandol Álvarez
aga@uci.cu, agacode@gmail.com
Universidad de las Ciencias Informáticas”

“Dr. Francisco Andrés Cano Alonso
fcano@uci.cu
Universidad de las Ciencias Informáticas”

AGRADECIMIENTOS

A mi abuela Eduarda Fernández por todo.

A mi madre por enseñarme a luchar por lo que creo y quiero y, por todos sus esfuerzos durante tantos años.

A mi padrastro Urbano Valier Dimas por ocuparse de nosotros.

A mis hermanos Héctor Valier Fernández y Eyliani Valier Fernández por estar conmigo cuando más los he necesitado.

A mi novia Idalmis Téllez Peña por apoyarme tanto.

A mi tutor Arnaldo Gandol por ayudarme con sus conocimientos.

A todos mis amigos del barrio por estar conmigo en las buenas y malas.

De nada sirve ser grande sin tener orgullo propio.

Anónimo.

RESUMEN

Este trabajo consiste en el diseño y desarrollo de un Módulo denominado salvar-cargar para el Simulador de la Industria Química Cubana. Este se encarga de salvar los Diagramas de Flujo de Información (DFI) realizados en el simulador sin alterar ningún dato a la hora de salvarlo y también se encarga de cargar estos DFI sin agregar un dato que no haya sido salvado, sin necesidad de contar con la misma versión, pues permite cargar DFI desarrollados en versiones anteriores a partir de una nueva propuesta para salvar y cargar. Para su diseño se utilizó el Lenguaje de Modelado UML sobre la herramienta Enterprise Architect [2], y para su desarrollo se utilizó el lenguaje C#.

Este trabajo de Diploma Productivo surge a partir de la necesidad de arreglar el Módulo salvar-cargar a partir de la versión anterior de este simulador, contando ya con una nueva arquitectura mejorada.

Con el desarrollo de este módulo se logrará el cumplimiento de las necesidades del cliente y un mejor proceso de salva y carga de manera confiable y cómoda por parte del usuario final.

Este trabajo consta de 3 capítulos. El primer capítulo trata sobre el *"Fundamento Teórico"* donde se abordará principalmente el estado del arte. El segundo es del rol *"Diseñador de Sistema"* y el tercero que es ya el *"Análisis de los resultados"*.

PALABRAS CLAVE

Serialización, Deserialización, Mever-framework, Shemaver, Adaver, Genever

ABSTRACT

This work consists on the design and development of a Module called save-charge for the Flight simulator of the Cuban Chemical Industry. This it takes charge of saving the Flow Charts of Information (DFI) carried out in the flight simulator without altering no data at the moment of to save it and also takes charge of charging these DFI without adding a data that have not been saved, without need to count on the same version, therefore permits to charge DFI developed in previous versions. For its design the Language was utilized of Shaped UML on the tool Enterprise Architect [2], and for its development the language was utilized C#.

This work of Productive Diploma arises from the need to fix the I Modulate save-charge from the previous version of this flight simulator, counting already with a new architecture improved.

With the development of this module the fulfilment of the needs of the client will be achieved and a better process of saves and charges in a dependable way and commode on the part of the end user.

This work is comprised of 3 chapters. The first chapter treats on the "Theoretical Base" where will be undertaken mainly the state of the art. The second is of the role "Designer of System" and the third that is already the "Analysis of the results".

INDICE

INTRODUCCION

El mundo avanza constantemente con el desarrollo de la Informática y los productos que salen al mercado, uno de esos productos son los simuladores que debido a su utilización en diversas industrias como centrales y plantas refinadoras de petróleo requieren de un gran esfuerzo para su desarrollo y puesta en funcionamiento, debido a esto en la Universidad de las Ciencias Informáticas(UCI) se desarrolló un simulador para la industria química cubana, denominado CheSys 1, para satisfacer la demanda de la industria azucarera y las docentes de varias Facultades de Ingeniería Química de las universidades cubanas.

Situación Problémica: El Módulo salvar-cargar de CheSys 1 no resuelve los problemas que se derivan de estas dos funciones, pues no permite que DFI desarrollados en una versión anterior se puedan cargar en una nueva. Además no permite que se salven los datos de forma personalizada o sea, solo los de interés para el desarrollador.

Problema: ¿Cómo diseñar un sistema con configuraciones que hagan totalmente independiente el módulo salvar-cargar del Sistema en general?

Objeto: Proceso de desarrollo del módulo salvar-cargar.

Objetivos:

- Desarrollar un módulo informático que permita delegar a un solo usuario el proceso de salvar y cargar.
- Desarrollar un módulo informático que permita cargar DFI desarrollados en versiones anteriores.

Campo de Acción: Proceso de desarrollo del módulo salvar-cargar para CheSys 2.

Hipótesis: Si se logra diseñar un buen sistema informático que salve-cargue DFI en cualquier versión de CheSys sin perder información se logrará entonces un sistema capaz de poder salvar-cargar la información con control de versiones.

Tareas de Investigación:

- Revisión del módulo salvar-cargar de CheSys 1.
- Revisión de los Requerimientos Funcionales Mínimos (RFM) firmados con los clientes.
- Modelación y diseño del módulo.
- Estudio de la arquitectura de CheSys 2.

Los métodos utilizados para el desarrollo de este trabajo dentro de los métodos teóricos fueron el método Analítico – sintético, el Inductivo – deductivo**,** el Análisis histórico lógico, el Genético y el de Modelación. Dentro de los empíricos utilizados encontramos la Entrevista y el Experimento.

A continuación se describen cada uno de estos métodos:

- **Analítico – sintético:** Son dos procesos inherentes al pensamiento, operaciones lógicas importantes; que nos permiten; como métodos teóricos, buscar la esencia de los fenómenos, los rasgos que lo caracterizan y los distinguen.

Su objetivo en una investigación es analizar las teorías, documentos, etc.; permitiendo la extracción de los elementos más importantes que se relacionan con el objeto de estudio.

- **Inductivo – deductivo:** Son formas de razonamiento que nos permite llegar a un grupo de conocimientos generalizadores, tanto desde el análisis de lo particular a lo general, como desde el análisis de elementos generalizadores a uno de menor nivel de generalización.

- **Análisis histórico lógico:** Nos permite estudiar de forma analítica la trayectoria histórica real de los fenómenos, su evolución y desarrollo.

Su objetivo en una investigación: constatar teóricamente como ha evolucionado un determinado fenómeno en un período de tiempo, en toda su trayectoria o en un fragmento temporal de la lógica de su desarrollo.

- **Genético:** Consiste en determinar el campo de acción del objeto a estudiar en la investigación, en la que están representados sus componentes, así como sus leyes más trascendentales.

El análisis del objeto, con un enfoque genético, permite deducir y explicar a partir de las leyes del comportamiento de la célula el desarrollo de sistemas más complejos.

- **Modelación:** Este método nos permite la creación de modelos, (propuestas, alternativas, estrategias, etc.)

El modelo es una reproducción simplificada de la realidad, cumple una función heurística, que permite descubrir y estudiar nuevas relaciones y cualidades del objeto de estudio. La modelación es justamente el proceso mediante el cual creamos modelos con vistas a investigar la realidad.

- **Entrevista:** Es una conversación planificada para obtener información.

Su uso constituye un medio para el conocimiento cualitativo de los fenómenos o sobre características personales del entrevistado y puede influir en determinados aspectos de la conducta humana por lo que es importante una buena comunicación.

- **Experimento:** Es el método empírico para el estudio de un objeto en el cual el investigador crea las condiciones o adapta las existentes para el esclarecimiento de las propiedades, leyes y relaciones del objeto, para verificar una hipótesis, una teoría o un modelo.

Estos métodos son importantes porque hacen factible seleccionar, acumular y realizar un análisis preliminar de la información obtenida pero, además, posibilitan verificar y comprobar las concepciones teóricas. Por ello no podemos hablar, en forma desvinculada, de investigación empírica e investigación teórica; la continua interacción de lo empírico y lo teórico conduce a la ampliación y el enriquecimiento del conocimiento científico.

Entre las principales etapas que tuvo esta investigación encontramos:

- Estudio del fenómeno.
- Análisis y diseño de este fenómeno.
- Implementación.

CAPITULO 1. FUNDAMENTO TEORICO.

1.1 Introducción.

Este capítulo trata acerca del estado actual de los diseños de sistemas desarrollados nacionales e internacionales, aborda además criterios de Pressman y RUP acerca del rol de diseñador de sistemas, también se abordarán las herramientas analizadas para el desarrollo de este trabajo.

1.2 Estado del Arte.

El diseño de sistemas es un arte, es más que la simple construcción [3], él envuelve más imaginación y demanda una perspectiva más amplia que el entorno del trabajo en programación, donde no están claras las especificaciones de entrada.

Este evoluciona con el avance de numerosos sistemas a medida que el desarrollo de la informática avanza también y con ello se ha vuelto cada vez mejor en la identificación de cada subsistema y que va cambiando continuamente al desarrollarse métodos nuevos, análisis mejores y se amplia el conocimiento.

La evolución del diseño de software es un proceso continuo que ha abarcado las últimas cuatro décadas. El primer trabajo de diseño se concentraba en criterios para el desarrollo de programas modulares y métodos para refinar las estructuras del software de manera descendente. Los aspectos procedimentales de la definición de diseño evolucionaron en la filosofía denominada *programación estructurada*. Un trabajo posterior propuso métodos para la conversión del flujo de datos o estructura de datos en la definición de diseño. Enfoques de diseño más recientes hacia la

derivación de diseño proponen un método orientado a objetos. Hoy en día, se ha hecho hincapié en un diseño de software basado en la arquitectura del software.

El diseño del software, al igual que los enfoques de diseño de ingeniería en otras disciplinas, va evolucionando a medida que se van desarrollando métodos nuevos de análisis mejores y se amplia el conocimiento. Las metodologías de diseño carecen de la profundidad, flexibilidad y naturaleza cuantitativa que se asocian normalmente a las disciplinas del diseño más clásicas.
Sin embargo, si existen métodos para el diseño del software como ejemplo, el diseño modular , que reduce la complejidad, facilita los cambios (un aspecto crítico de la capacidad de mantenimiento del software), ya da como resultado una implementación más fácil de fomentar el desarrollo paralelo de las diferentes partes de un sistema.

Otro método del diseño es el arquitectónico, que se encarga de representar la estructura de los datos o los componentes de un programa que se requiere para construir un sistema basado en computadora. Constituye el estilo arquitectónico que tendrá el sistema, la estructura y las propiedades de los componentes que ese sistema comprende y las interrelaciones que tienen lugar entre todos los componentes arquitectónicos del sistema.

También existe el diseño a nivel de componentes que tiene lugar después de haber establecido los diseños de datos, de interfaces y de arquitectura que su objetivo es convertir el modelo del diseño en un software operacional .Esta conversión puede ser un desafío, abriendo puertas a la introducción de errores sutiles que sean difíciles de detectar y corregir en etapas posteriores del proceso del software.

Para terminar con los métodos se habla del diseño orientado a objetos (DOO), este diseño transforma el modelo del análisis orientado a objetos, en un modelo de diseño que sirve como anteproyecto para la construcción del software. El trabajo del diseñador puede ser intimidante. Este tipo de diseño es difícil y el diseño de software reusable es más difícil aún.

1.3 Criterios de Autores.

1.3.1 Pressman.

Según Pressman [3] el objetivo de los diseñadores es producir un modelo o representación de una entidad que será construida a posteriori. En el contexto de la ingeniería de software el diseño se centra en cuatro áreas importantes de interés: datos, arquitectura, interfaces y componentes. Su seguimiento se puede hacer basándose en los requisitos del cliente, y al mismo tiempo la calidad se puede evaluar.

Independientemente del modelo de diseño que se utilice el ingeniero de software deberá aplicar un conjunto de principios fundamentales y principios básicos para el diseño a nivel de componentes, de interfaz, arquitectura y datos. Estos principios son:

- En el proceso de diseño no deberá utilizarse << orejeras>>.
- El diseño deberá poderse rastrear hasta el modelo de análisis.
- El diseño no deberá inventar nada que ya esté inventado.
- El diseño deberá <<minimizar la distancia intelectual>> entre el software y el problema como si de la misma vida real se tratara.
- El diseño deberá presentar uniformidad e integración.
- El diseño deberá estructurarse para admitir cambios.
- El diseño deberá estructurarse para degradarse poco a poco, incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes.
- El diseño no es escribir código y escribir código no es diseñar.
- El diseño deberá evaluarse en función de la calidad mientras se va creando.
- El diseño deberá revisarse para minimizar los errores conceptuales (semánticos).

1.3.2 RUP.

Según RUP [4] el diseño debe:

-Ejecutar las tareas y funciones descritas en los casos de uso.

-Satisfacer todos los requerimientos.

-Flexible a cambios.

Además debe centrarse en la noción de arquitectura y la validación de esta se convierte en una tarea esencial.

RUP plantea [4] que el modelo de diseño consta de:

- Clases estructuradas en paquetes.
- Diseños de subsistemas con interfaces definidas (componentes).
- Forma de colaboración entre las clases.

De acuerdo a ambos criterios la decisión estuvo dada por lo que plantea RUP porque se hizo un seguimiento de cada uno de los Casos de Uso, que está representado por los requerimientos planteados por los clientes y siempre con una visión flexible a cambios en esos requerimientos y de manera que este diseño sea los más ajustado al análisis realizado, pero siempre con una buena robustez en su diseño.

1.4 Herramientas para el diseño y desarrollo del sistema.

Las herramientas para el diseño de sistemas apoyan el proceso de formular las características que el sistema debe tener para satisfacer los requerimientos detectados durante las actividades del análisis. Estas herramientas son:

1.4.1 .NET.

¿Qué es .NET?

Es la denominación de Microsoft para su nueva arquitectura de tecnologías como: el .NET [14] Framework, los lenguajes de programación .NET (C#, VB, C++, J#), herramientas y servidores .NET.

Microsoft .NET [14] es un programa de software que conecta información, usuarios, sistemas y dispositivos. Incluye clientes, servidores y herramientas para programadores, y está formado por:

1. Windows .NET Framework que permite generar y ejecutar todo tipo de software, incluidas aplicaciones basadas en Web, aplicaciones cliente inteligentes y servicios Web XML. Estos componentes facilitan la integración, ya que comparten datos y funcionalidad a través de una red mediante protocolos estándar independientes de la plataforma, como XML, SOAP y HTTP.
2. Varias herramientas para programadores, como Microsoft Visual Studio .NET 2003, que ofrece un entorno de desarrollo integrado (IDE) para sacar el máximo partido a la productividad de los programadores con Windows .NET Framework.
3. Un conjunto de servidores, incluidos Microsoft Windows Server 2003, Microsoft SQL Server y Microsoft BizTalk Server, que integran, ejecutan, operan y administran servicios Web XML y aplicaciones basadas en Web.
4. Software cliente, como Windows XP, Windows CE o Microsoft Office XP, que ayuda a los programadores a ofrecer una experiencia positiva para el usuario a través de la amplia familia de dispositivos y productos existentes.

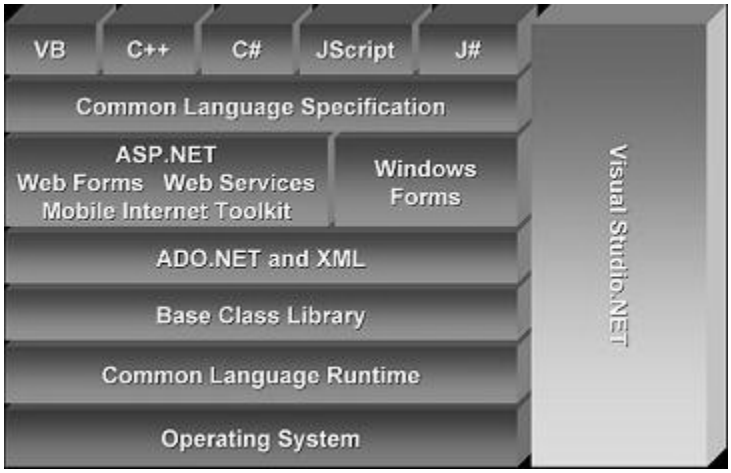


Ilustración 1. Arquitectura de .NET Framework

Los puntos fuertes de la plataforma son:

- **Independencia de lenguaje**

Todos los lenguajes que conformen con los estándares .NET, sin importar cual, podrán interoperar entre sí de forma totalmente transparente, las clases podrán ser heredadas entre unos lenguajes y otros, y se podrá disfrutar de polimorfismo entre lenguajes. Por ejemplo, si yo tengo una clase en C#, esta clase podrá ser heredada y utilizada en Visual Basic o JScript o cualquier lenguaje .NET. Todo esto es posible por medio de una de las características de .NET llamado Common Type System (CTS).

También tiene la cualidad de que se pueden incluir más lenguajes a la plataforma. En la actualidad existen proyectos independientes de incluir PHP, Python, Ada y otros lenguajes en la plataforma.

- **Librería de clases común:**

Más de 4000 clases, objetos y métodos incluidos en la plataforma .NET están disponibles para todos los lenguajes.

- **Multiplataforma**

Cuando un programa es compilado, no es compilado en un archivo ejecutable sino en un lenguaje intermedio llamado “Lenguaje Intermedio” (IL) el cual podrá ser ejecutado por el CLR (Common Language Runtime) en la plataforma en que el CLR esté disponible. (Hasta el día de hoy Microsoft solamente tiene un CLR para los sistemas operativos Windows, pero el proyecto Mono y dotGNU han puesto a disposición un CLR para GNU/Linux, MacOS y muchas otras plataformas).

Los sistemas operativos Windows XP o superiores incluyen el CLR nativamente y SuSE Linux 9.3 o superior planea incorporar el CLR (Mono) en su distribución lo que quiere decir que un programa .NET podrá ser compilado y ejecutado en cualquiera de estas plataformas, o en cualquier plataforma que incluya un CLR.

El CLR compilará estos archivos IL nuevamente en código de máquina en un proceso que se conoce como JIT (justo a tiempo) el cual se ejecutará cuando se requiera. Este proceso producirá código de máquina bien eficiente que se reutilizará si es que hubiera código que se repitiera, haciendo que los programas sean ejecutados muy eficientemente.

- **Windows Forms, Web Forms, Web Services**

La plataforma .NET incluye un conjunto de clases especiales para datos y XML que son la base de 3 tecnologías claves: Servicios Web (Web Services), Web Forms, y Windows Forms los cuales son poderosas herramientas para la creación de aplicaciones tanto para la plataforma como para la Web.

- **Estandarización**

Además de los méritos técnicos, una de las razones del éxito de la plataforma .NET ha sido por el proceso de estandarización que Microsoft ha seguido (y que ha sorprendido a más de uno). Microsoft, en lugar de reservarse todos los derechos sobre el lenguaje y la plataforma, ha publicado las especificaciones del lenguaje y de la plataforma, que han sido posteriormente revisadas y ratificadas por la Asociación Europea de Fabricantes de Computadoras (ECMA).
Esta especificación (que se puede descargar libremente de Internet) permite la implementación del lenguaje C# y de la plataforma .NET por terceros, incluso en entornos distintos de Windows.

¿Qué es el .NET Framework?

Es la plataforma y componente principal de .NET, el cual brinda un conjunto de nuevas tecnologías y herramientas que nos proporcionan todo lo necesario para construir nuestras aplicaciones de diversos tipos ya sean: aplicaciones de consola, de Windows, Web Services, librerías de clases, aplicaciones Web, aplicaciones para dispositivos inteligentes y otros. El .NET Framework está basado en estándares, básicamente en: HTTP, XML y SOAP lo que le permite una fácil interoperabilidad y migración a otras plataformas.

Microsoft perseguía dos objetivos principales en la creación de la plataforma .NET. El primero era mejorar el desarrollo de Microsoft Windows haciéndolo más orientado a objetos y simplificando el modelo de objetos subyacente. El segundo era ofrecer un marco de trabajo moderno de tercera generación para crear e implementar servicios Web XML. Con .NET Framework, Microsoft mejora enormemente la productividad del programador, la facilidad de desarrollo y la ejecución de aplicaciones confiables. Los servicios Web XML integran aplicaciones y componentes poco complementados diseñados para el heterogéneo entorno informático actual mediante la comunicación con protocolos de Internet estándar como SOAP, WSDL (Lenguaje de descripción de servicios Web) y UDDI (Integración, descubrimiento y descripción universal).

.NET Framework consta de tres áreas principales. La primera es Common Language Runtime (CRL), que asume la responsabilidad de ejecutar la aplicación. CRL garantiza que se cumplan todas las dependencias de la aplicación, administra la memoria y controla cuestiones como la seguridad y la integración de lenguajes. Common Language Runtime proporciona muchos servicios que simplifican el desarrollo del código y la implementación de la aplicación a la vez que aumenta la fiabilidad de la aplicación. La mayor parte de este trabajo se controla de manera transparente, simplificando las tareas de los desarrolladores y administradores.

La segunda área es la de las clases principales unificadas. Estas clases proporcionan todos los recursos que requiere un desarrollador para generar una aplicación moderna, incluida la compatibilidad con XML, las conexiones de red y el acceso a datos. Tener estas clases unificadas significa que un desarrollador que desarrolle cualquier tipo de aplicación, basada en Windows o en Web, utiliza las mismas clases. Esta coherencia aumenta la productividad del desarrollador y la reutilización de código.

La tercera y última área es la de las clases de presentación, que incluyen ASP.NET para el desarrollo de aplicaciones Web, así como servicios Web XML y Windows Forms para el desarrollo de aplicaciones basadas en Windows o de "cliente inteligente".

La plataforma .NET no son sólo los servicios Web, sino que también ofrece numerosos servicios a las aplicaciones que para ella se escriban, como son un recolección de basura, independencia de la plataforma, total integración entre lenguajes (por ejemplo, es posible escribir una clase en C# que derive de otra escrita en Visual Basic.NET que a su vez derive de otra escrita en Cobol).

El lenguaje de programación C#.

C# [15] es un lenguaje de programación moderno, se podría decir que es tanto una evolución de los lenguajes C++ y Java, ya que incorpora las mejores características de cada uno de ellos. C# es un lenguaje muy estricto en su sintaxis ya que obliga a que se declaren todas las variables, las

conversiones de datos deben ser explícitas, es sensible a mayúsculas y otras características que lo convierte en un lenguaje muy formal. Esto no sucede en otros lenguajes al menos de forma inicial como en Visual Basic .NET, y aunque esta formalidad puede ser una dificultad al comienzo a la larga ayuda al desarrollador a su formación ya que le proporciona mayor disciplina.

Es un lenguaje orientado a objetos simple, seguro, moderno, de alto rendimiento y con especial énfasis en Internet y sus estándares (como XML). Es también la principal herramienta para programar en la plataforma .NET.

C# combina los mejores elementos de múltiples lenguajes de amplia difusión como C++, Java, Visual Basic o Delphi. De hecho, su creador Anders Heljsberg fue también el creador de muchos otros lenguajes y entornos como Turbo Pascal, Delphi o Visual J++. La idea principal detrás del lenguaje es combinar la potencia de lenguajes como C++ con la sencillez de lenguajes como Visual Basic, y que además la migración a este lenguaje por los programadores de C/C++/Java sea lo más inmediata posible.

El lenguaje es muy sencillo, sigue el mismo patrón de los lenguajes de programación modernos. Incluye un amplio soporte de estructuras, componentes, programación orientada a objetos, manipulación de errores, recolección de basura, etc., que es construido sobre los principios de C++ y Java. Como sabréis, las clases son la base de los lenguajes de programación orientados a objetos, lo cual permite extender el lenguaje a un mejor modelo para solucionar problemas. C# contiene las herramientas para definir nuevas clases, sus métodos y propiedades, al igual que la sencilla habilidad para implementar encapsulación, herencia y polimorfismo, que son los tres pilares de la programación orientada a objetos. C# tiene un nuevo estilo de documentación XML que se incorpora a lo largo de la aplicación, lo que simplifica la documentación en línea de clases y métodos. C# soporta también interfaces, una forma de estipular los servicios requeridos de una clase. Las clases en C# pueden heredar de un padre pero puede implementar varias interfaces.

C# también provee soporte para estructuras, un concepto el cual ha cambiado signifivamente desde C++. Una estructura es un tipo restringido que no exige tanto del sistema operativo como una clase. Una estructura no puede heredar ni dar herencias de clases pero puede implementar una interface. C# provee características de componentes orientados, como propiedades, eventos y construcciones declaradas (también llamados atributos). La programación orientada a componentes es soportada por el CLR. C# provee soporte para acceder directamente a la memoria usando el estilo de punteros de C++ y mucho más.

¿Por qué C#?

La plataforma .NET acepta varios lenguajes. Por ahora, C#, Visual Basic, C++ gestionado, Nemerle, FORTRAN, Java, Python, etc., y con capacidad para aceptar prácticamente cualquier lenguaje. Entonces la pregunta es, ¿porqué se eligió C# en lugar de cualquier otro lenguaje?

La razón fundamental es que C# se diseñó para la plataforma .NET y es capaz de utilizar todo su potencial. También es cierto que es un lenguaje "limpio" en el sentido de que al no tener que proporcionar compatibilidad hacia detrás se ha tenido más libertad en el diseño y se ha puesto especial hincapié en la simplicidad. Por ejemplo, en C# hay un tipo de clase y siempre se le aplica el recolector de basura mientras que en C++ gestionado hay dos tipos de clases, una a la que se aplica el recolector y otra a la que no.

1.4.2 RUP.

Se hizo uso de las herramientas de la metodología RUP [4] (*Rational Unified Process*) para facilitar el desarrollo del sistema.

El Proceso Unificado es un proceso de desarrollo de software que contiene un conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema de software. Más que un simple proceso; es un marco de trabajo genérico que puede especializarse para una

gran variedad de sistemas software, para diferentes áreas de aplicación, tipos de organizaciones, niveles de actitud y tamaños de proyecto. Está basado en componentes, lo cuál quiere decir que el sistema software en construcción está formado por componentes software interconectados a través de interfaces bien definidas.

Utiliza el *Lenguaje Unificado de Modelado* (Unified Modeling Languaje, UML) para visualizar, especificar, construir y documentar los artefactos de un sistema que involucra una gran cantidad de software. Está compuesto por diversos elementos gráficos que se combinan para conformar diagramas. Es importante recalcar que UML no es una guía para realizar el análisis y diseño orientado a objetos, es decir, no es un proceso. UML es un lenguaje que permite la modelación de sistemas con tecnología orientada a objetos.

Características principales de RUP:

1. **Dirigido por casos de uso:** Los casos de uso reflejan lo que los usuarios futuros necesitan y desean, lo cual se capta cuando se modela el negocio y se representa a través de los requerimientos. A partir de aquí los casos de uso guían el proceso de desarrollo ya que los modelos que se obtienen, como resultado de los diferentes flujos de trabajo, representan la realización de los casos de uso (cómo se llevan a cabo).

2. **Centrado en la arquitectura:** La arquitectura muestra la visión común del sistema completo en la que el equipo de proyecto y los usuarios deben estar de acuerdo, por lo que describe los elementos del modelo que son más importantes para su construcción, los cimientos del sistema que son necesarios como base para comprenderlo, desarrollarlo y producirlo económicamente. RUP se desarrolla mediante iteraciones, comenzando por los CU relevantes desde el punto de vista de la arquitectura.

3. **Iterativo e Incremental:** RUP propone que cada fase se desarrolle en iteraciones. Una iteración involucra actividades de todos los flujos de trabajo, aunque desarrolla fundamentalmente algunos más que otros. Por ejemplo, una iteración de elaboración centra su atención en el análisis y diseño, aunque refina los requerimientos y obtiene un producto con un determinado nivel, pero que irá creciendo incrementalmente en cada iteración. Es práctico dividir el trabajo en partes más pequeñas o miniproyectos. Cada miniproyecto es una iteración que resulta en un incremento. Las iteraciones hacen referencia a pasos en los flujos de trabajo, y los incrementos, al crecimiento del producto. Cada iteración se realiza de forma planificada es por eso que se dice que son miniproyectos.

1.4.3 Enterprise Architect.

Enterprise Architect [2] 4.5 (EA) es una herramienta flexible, completa y potente de modelado en UML bajo plataforma Windows. Provee lo más nuevo en desarrollo de sistemas, administración de proyectos y análisis de negocio.

EA es una herramienta que abarca integralmente el ciclo de vida, cubriendo el desarrollo de software desde el levantamiento de los requerimientos, a través de las etapas de análisis, modelos de diseño, testing y finalmente el mantenimiento y re-uso.

EA es utilizada para el desarrollo de varios tipos de software para un amplio rango de industrias, incluyendo: bancos, desarrollo web, ingeniería, finanzas, medicina, investigación, educación, transporte, ventas, energía, ingeniería electrónica y muchas más. También es utilizado con efectividad para el entrenamiento en UML y arquitecturas de negocio en empresas de entrenamiento y universidades alrededor del mundo.

Enterprise Architect 6.5 fue construido en base al excepcional éxito de las versiones previas con un completo soporte par el estándar UML 2.1 como lo ha definido la OMG. Con EA 6.5, los modeladores tienen todo el poder y la expresividad de los 13 diagramas de UML 2.1 en sus manos, incluyendo:

Diagramas de Estructura:	Diagramas de Comportamiento:	Extensiones Temporales:
Clases Objetos Compuesto Paquetes Componentes Despliegue	Casos de Uso Comunicación Secuencia Interacción Actividad Estado	Análisis Personalizados (requisitos, diseño de UI)

1.4.4 Reflexión.

Los servicios de reflexión [5] permiten, a través de los metadatos, obtener en tiempo de ejecución información sobre los tipos definidos en una aplicación o de los módulos a los que acceda. De esta forma, puede obtenerse dinámicamente el nombre del ensamblado que se está ejecutando, interrogar a distintas clases por los métodos que ofrecen, los campos y propiedades públicos, etc. Para ello, mediante los servicios ofrecidos por el espacio de nombres System.Reflection puede obtenerse el tipo de cualquier instancia e interrogarlo para obtener todas sus propiedades.

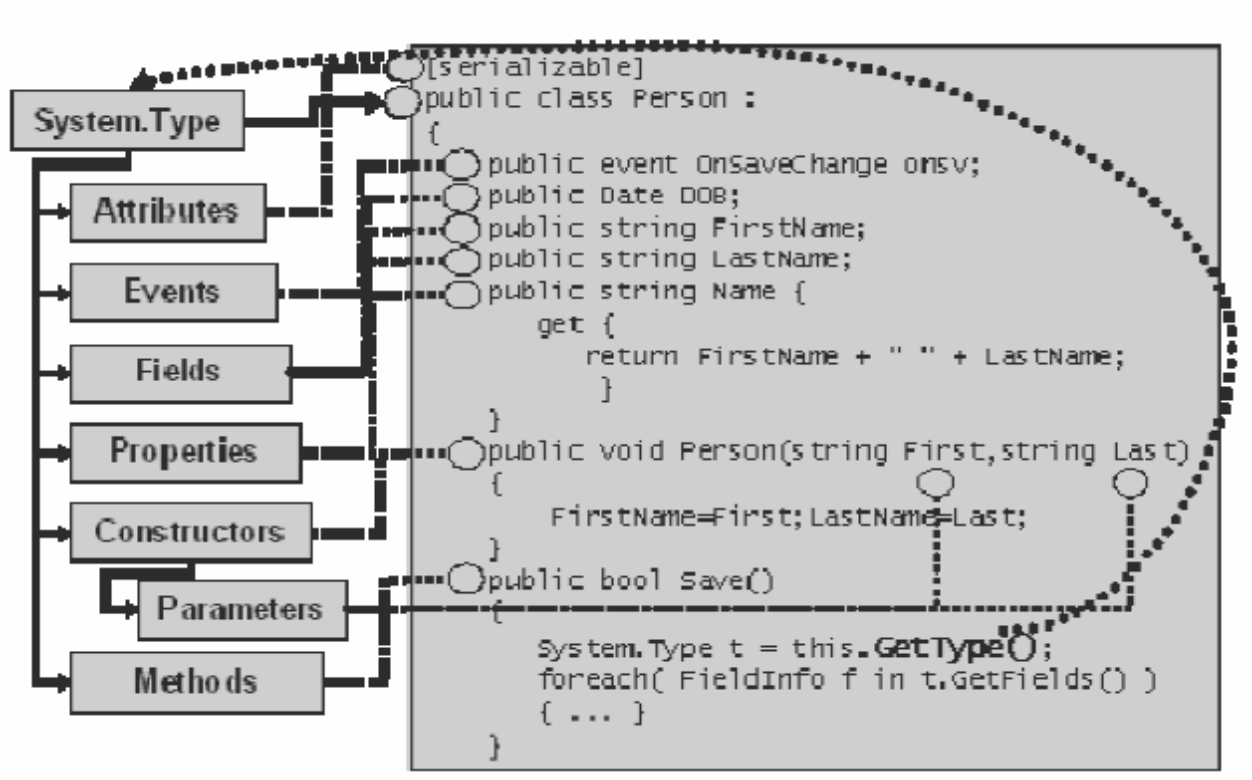


Ilustración 2. Componentes de Reflection.

La capacidad de reflexión de la plataforma .NET (similar a la de la plataforma Java) nos permite explorar información sobre los tipos de los objetos en tiempo de ejecución.

Una primera aplicación que se puede derivar de este mecanismo es el “enlace tardío” (o Late Binding) mediante el cual las librerías dinámicas se pueden cargar y enlazar dinámicamente a la aplicación en función de si contienen clases específicas requeridas por la aplicación (por ejemplo, que herede de una determinada clase, o que implementen determinada interfaz). Además, la reflexión también es utilizada por otras librerías de .Net con diversos fines: por ejemplo para serializar objetos (guardar el estado de un objeto que se encuentra en memoria) se utiliza la reflexión para obtener los campos que pueden ser serializados o no.

Sin embargo, la principal ventaja de los servicios de reflexión proporcionados por la librería es la posibilidad de emitir dinámicamente código. Para ello se utilizan los tipos y sus correspondientes servicios definidos en el espacio de nombres System.Reflection.Emit para crear primero un ensamblado en memoria, definir los tipos que lo formarán (junto a sus miembros) y proceder a emitir instrucciones MSIL.

Posteriormente, el ensamblado podrá guardarse en disco o dejarse en memoria si no se necesita la persistencia del código generado. El principal inconveniente que plantea este mecanismo es que requiere por parte del desarrollador un conocimiento profundo del código intermedio, a medio camino entre los lenguajes de alto nivel y el lenguaje máquina, lo que puede provocar numerosos errores de codificación.

1.5 Serialización/Deserialización.

En ciencias de la computación, la serialización [6] (o marshalling en inglés) consiste en un proceso de codificación de un Objeto (programación orientada a objetos) en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como XML. La serie de bytes o el formato pueden ser usados para re-crear un objeto que es idéntico en su estado interno al objeto original (actualmente un clon). La serialización es un mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones o localizaciones.

NET Framework soporta dos tipos de serialización, poco profundo y profunda [9]. La Serialización poco profundo es el proceso de convertir el valor de las property que sean de lectura y escritura de un objeto, en un flujo de byte, esta es la técnica usada por la clase XmlSerializer [8] y

Servicios de Web. Este proceso se llama serialización poco profunda, porque no serializa el estado completo de un objeto, sólo los estados disponibles.

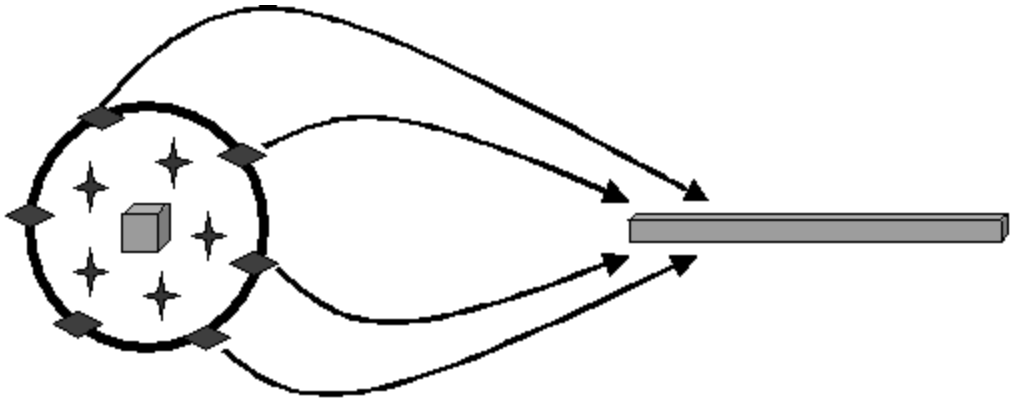


Ilustración 3. Serialización, poco profunda —proceso de copiar el valor de las property a un flujo de bytes.

La Serialización profunda [8] es el proceso de convertir los valores reales guardados en las variables de un objeto en un flujo bytes. Es la técnica usada por las clases BinaryFormatter y SoapFormatter, y por .NET Remoting. También se usa de una forma más limitada para serializar los datos guardados en las páginas Web. La Serialización profunda es más completa, porque se pueden copiar los valores que se guardan en las variables privadas. Proporciona una copia mucho más completa de los estados del objeto.

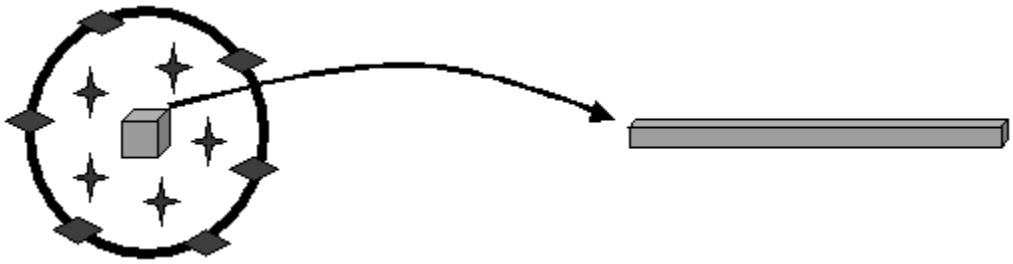


Ilustración 4. Serialización profunda —proceso de copiar el valor de un objeto a un flujo de bytes.

Adicionalmente, la Serialización profunda fabricará en serie un grafo entero del objeto. En otros términos, si su objeto sostiene una referencia a otro objeto, o a una colección de otros objetos, todos esos objetos serán incluidos también en el proceso del serialización.

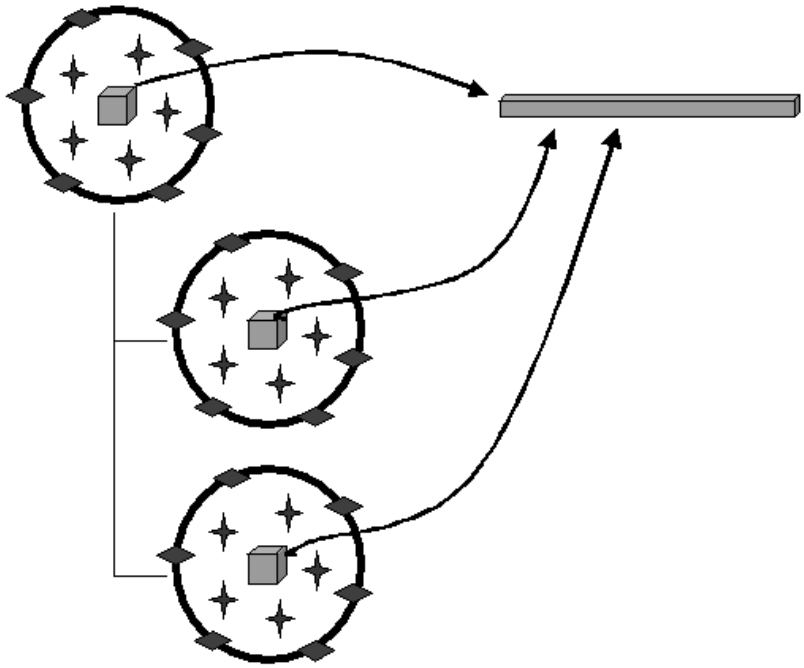


Ilustración 5. Serializando el grafo de un objeto —muchos objetos combinados dentro del flujo de bytes.

Durante este proceso, los campos públicos y privados del objeto y el nombre de la clase, incluidos el ensamblador que contiene la clase, se convierten en una secuencia de bytes que, posteriormente, se escribirá en una secuencia de datos. Más adelante, cuando se deserializa el objeto, se creará un clon exacto del objeto original.

Existen 4 métodos para la serialización pero solo uno es el ideal para poder tener control de versiones sobre los ficheros, y permitir referencias cíclicas entre entidades. XML serializa en un formato bien fácil de entender, pero tiene muchas restricciones solo puede serializar las

propiedades que sean públicas y sean de lectura y de escritura, además no permite referencias cíclicas y este es un punto clave en los DFI que se modelan en CheSys.

Soap y Binary son ideales por que no tienen tantas restricciones como XML, pero no suficientes, a menos que no nos interese tener un control profundo sobre el proceso de serialización. Custom es el ideal, permite hacer una serialización profunda, tener referencias cíclicas y además un control profundo de la serialización de hecho el mecanismo que se utiliza para resolver el problema de control de versiones se resuelve a través de este tipo de serialización.

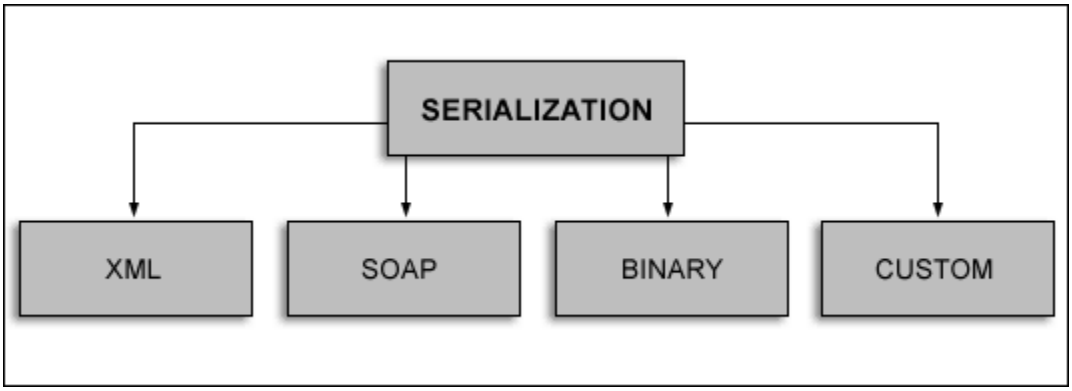


Ilustración 6. Tipos de serialización en .Net framework

Se propone un framework para salvar y cargar los DFI que se modelan en CheSys, en un fichero serializado, que pueda intercambiar información con ficheros de futuras versiones, que se puedan guardar los datos necesarios y que el programador no tenga que preocuparse por los detalles de este proceso, con el fin de evitar posibles entradas de errores sutiles.

Cuando se implementa un mecanismo de serialización en un entorno orientado a objetos, es preciso crear un equilibrio entre la facilidad de uso y la flexibilidad. A su vez, este proceso se podrá automatizar aún más, si tiene suficiente control sobre éste. Por ejemplo, puede ocurrir que una serialización binaria no sea suficiente o exista una razón concreta para decidir qué campos

deben serializarse en una clase. Seguidamente se destacan varias características importantes que permiten personalizar el proceso para que se adapte a sus necesidades, ellas son:

- **Almacenamiento persistente.** [9]

Normalmente, es necesario almacenar el valor de los campos de un objeto en un disco para poder recuperar los datos posteriormente. Si bien esto se consigue sin la serialización, puede resultar un proceso problemático y dar lugar a errores, e incluso complicarse si se necesita realizar un seguimiento de la jerarquía de los objetos. Imagine que debe escribir una aplicación para una gran empresa que contenga varios miles de objetos y tiene que escribir código para guardar en un disco y recuperar desde éste los campos y propiedades para cada objeto. La serialización proporciona el mecanismo adecuado para conseguir este objetivo con el mínimo esfuerzo.

El tiempo de ejecución en lenguaje común (CLR) administra la forma en que los objetos se exponen en la memoria y .NET Framework ofrece un mecanismo de serialización automática por reflexión. Cuando se serializa un objeto, se escriben el nombre de la clase, el ensamblado y todos los miembros de datos de la instancia de la clase para el almacenamiento. Con frecuencia, los objetos almacenan referencias a otras instancias en variables miembro.

Cuando se serializa la clase, el motor de serialización realiza un seguimiento de todos los objetos mencionados ya serializados para asegurar que el mismo objeto no se serialice más de una vez. La arquitectura de serialización en .NET Framework controla gráficos de objetos y referencias circulares automáticamente. El único requisito que deben cumplir los gráficos de objeto es que todos los objetos mencionados por el objeto que está siendo serializado deben marcarse como serializable. De lo contrario, se enviará una excepción cuando el serializador intente serializar un objeto no marcado.

Al deserializar la clase serializada, se recrea la clase y los valores de todos los miembros de datos se recuperan automáticamente. La Deserialización es el proceso mediante el cual se recoge la información almacenada y se vuelven a crear objetos a partir de la misma.

- **Control de versiones.**

.NET Framework admite el control de versiones y la ejecución página tras página. Todas las clases funcionarán en las distintas versiones si no se cambian las interfaces. Dado que las serializaciones tratan con variables miembro y no interfaces. Éste es el caso para clases que no implementan **ISerializable**.

Cualquier cambio de estado en la versión actual, como la agregación de variables miembro, el cambio del tipo de variables o de sus nombres, significará que los objetos existentes del mismo tipo no podrán deserializarse correctamente si se serializaron en una versión anterior.

Si el estado de un objeto requiere cambiar versiones, los autores de clase tienen dos posibilidades:

- Implementar **ISerializable**, lo que permite tomar un control preciso de los procesos de serialización y deserialización. Como resultado, se podrá agregar e interpretar correctamente el estado futuro durante la deserialización.

- Marcar las variables miembro no esenciales con el atributo **NonSerialized**. Esta opción sólo debe utilizarse cuando se esperen cambios mínimos entre las diferentes versiones de una clase. Por ejemplo, al agregar una nueva variable a una versión posterior de una clase, la variable no puede marcarse como **NonSerialized** con el fin de asegurar que se mantenga compatible con las versiones anteriores.

- **Soporte en los lenguajes de programación.**

Varios lenguajes de programación orientados a objeto soportan la serialización de forma directa. Algunos de ellos son Objective-C, Java, C#, Visual Basic .NET, ColdFusion, Ocaml,Perl, C++, Python y PHP.

1.6 Diseño con Patrones y Patrones de diseño.

El software ha ido aumentando su complejidad de forma pareja a la potencia de las computadoras, el aumento del conocimiento de los usuarios, incremento de las posibilidades de la informática, de las nuevas posibilidades que ofrecen las redes e Internet, etc. Los patrones de diseño [1] proponen una forma reutilizar la experiencia de los desarrolladores, para ello clasifica y describe formas de solucionar problemas que ocurren de forma frecuente en el desarrollo. Por tanto está basado en la recopilación del conocimiento de los expertos en desarrollo de software.

Los patrones son soluciones de sentido común que deberían formar parte del conocimiento de un diseñador experto. Además facilitan la comunicación entre diseñadores, pues establecen un marco de referencia (terminología, justificación), nos hablan de como construir software, de como utilizar las clases y los objetos de forma conocida. Es necesario tener conocimientos previos de Programación Orientada a Objetos para entender los Patrones, es conveniente también tener al menos nociones de UML para seguir los ejemplos y diagramas que aquí se presentan.

1.6.1 Clasificación de los patrones.

- **Patrones de arquitectura:** se expresa una organización o esquema estructural fundamental para sistemas software. Proporciona un conjunto de subsistemas predefinidos y sus responsabilidades.
- **Patrones de diseño**: proporciona esquemas para refinar subsistemas o componentes de un sistema.
- **Patrones de programación**: Describe la implementación de aspectos de componentes.

- **Patrones de análisis**: prácticas que aseguran la consecución de un buen modelo de un problema y su solución.
- **Patrones organizacionales**: describen la estructura y prácticas de las organizaciones humanas.

1.6.2 Facilidades de los patrones de diseño.

- Facilitan la comunicación interna.
- Ahorran tiempo y experimentos inútiles.
- Mejoran la calidad del diseño y la implementación.
- Son como “normas de productividad”.

Los patrones de diseño [1] hacen más fácil reutilizar con éxito los diseños y arquitecturas, ayudan a elegir entre diseños alternativos, hacen a un sistema reutilizable y evitan alternativas que comprometen la reutilización.

Un patrón de diseño tiene cuatro elementos característicos:
El nombre del patrón, describe el problema de diseño, su solución, y consecuencias en una o dos palabras. Tener un vocabulario de patrones nos permite hablar sobre ellos.
El problema describe cuando aplicar el patrón. Se explica el problema y su contexto. Puede describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. Se incluye una lista de condiciones.
La solución describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones. No se describe un diseño particular. Un patrón es una plantilla.
Las consecuencias son los resultados de aplicar el patrón.

1.6.3 Cualidades de un patrón de diseño.

- **Encapsulamiento y abstracción.** Cada patrón encapsula un problema bien definido y su solución en un dominio particular.
- **Extensión y variabilidad.** Cada patrón debería ser abierto por extensión o parametrización por otros patrones, de tal forma que pueden aplicarse juntos para solución un gran problema.
- **Generatividad y composición.** Cada patrón una vez aplicado genera un contexto resultante, el que concuerda con el contexto inicial de uno o más de uno de los patrones del catálogo.
- **Equilibrio.** Cada patrón debe realizar algún tipo de balance entre sus efectos y restricciones.

1.6.4 Tipos de patrones de diseño.

- **Patrones de Creación**

Los patrones de creación abstraen la forma en que se crean los objetos, de forma que permite tratar las clases a crear de forma genérica apartando la decisión de qué clases crear o como crearlas.

Según donde se tome dicha decisión podemos clasificar a los patrones de creación en *patrones de creación de clase* (la decisión se toma en los constructores de las clases y usan la herencia para determinar la creación de las instancias) y *patrones de creación de objeto (se modifica la clase desde el objeto)*.

- **Patrones Estructurales**

Tratan de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos. Lo fundamental son las relaciones de uso entre los objetos, y, éstas están determinadas por las interfaces que soportan los objetos. Estudian como se relacionan los objetos en tiempo de ejecución. Sirven para diseñar las interconexiones entre los objetos.

- **Patrones de comportamiento**

Los patrones de comportamiento hablan de como interaccionan entre si los objetos para conseguir ciertos resultados.

Patrones de diseño (GoF)

Creación:	**Estructurales**:	**Comportamiento**:	
Abstract Factory	Adapter	Chain of Responsibility	State
Builder	Bridge	Command	Strategy
Factory Method	Composite	Interpreter	Template Method
Prototype	Decorator	Iterator	Visitor
Singleton	Facade	Mediator	
	Flyweight	Memento	
	Proxy	Observer	

1.6.4.1 Patrones de creación. Descripción.

- **Abstract Factory:** Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar su clase concreta.
- **Builder:** Permite a un objeto construir un objeto complejo especificando sólo su tipo y contenido.
- **Factory Method:** Define una interfaz para crear un objeto, dejando a las subclases decidir el tipo específico.
- **Prototype:** Permite a un objeto crear objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos.
- **Singleton:** Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él.

1.6.4.2 Patrones estructurales. Descripción.

- **Adapter**: convierte la interfaz que ofrece una clase en otra esperada por los clientes
- **Bridge**: desacopla una abstracción de su implementación y les permite variar independientemente.
- **Composite**: permite construir objetos complejos mediante composición recursiva de objetos similares.
- **Decorator**: extiende la funcionalidad de un objeto dinámicamente de tal modo que es transparente a sus clientes.
- **Facade**: simplifica los accesos a un conjunto de objetos relacionados proporcionando un objeto de comunicación.
- **Flyweight**: usa la compartición para dar soporte a un gran número de objetos de grano fino de forma eficiente.
- **Proxy:** proporciona un objeto con el que controlamos el acceso a otro objeto.

1.6.4.3 Patrones de comportamiento. Descripción.

- **Chain of Responsability:** evita el acoplamiento entre quien envía una petición y el receptor de la misma.
- **Command**: encapsula una petición de un comando como un objeto.
- I**nterpreter**: dado un lenguaje define una representación para su gramática y permite interpretar sus sentencias.
- **Iterator**: acceso secuencial a los elementos de una colección.
- **Mediator**: define una comunicación simplificada entre clases.
- **Memento**: captura y restaura un estado interno de un objeto
- **Observer**: una forma de notificar cambios a diferentes clases dependientes.
- **State**: modifica el comportamiento de un objeto cuando su estado interno cambia.

- **Strategy**: define una familia de algoritmos, encapsula cada uno y los hace intercambiables.
- **Template Method**: define un esqueleto de algoritmo y delega partes concretas de un algoritmo a las subclases.
- **Visitor**: representa una operación que será realizada sobre los elementos de una estructura de objetos, permitiendo definir nuevas operaciones sin cambiar las clases de los elementos sobre los que opera.

1.6.5 Patrones de Asignación de Responsabilidades (Grasp).

Un sistema orientado a objetos se compone de objetos que envían mensajes a otros objetos [11] para que lleven a cabo operaciones, los diagramas de interacción describen gráficamente la solución a partir de los objetos en interacción que estas responsabilidades y precondiciones satisfacen.

Grasp es un acrónico que significa **G**eneral **R**esponsibility **A**signmet **S**oftware **P**atterns [12]. Estos patrones se aplican durante la elaboración de los diagramas de interacción, al asignar las responsabilidades a los objetos y al diseñar las colaboraciones entre ellos.

Patrones Básicos

- Experto.
- Creador.
- Bajo Acoplamiento.
- Alta Cohesión.
- Controlador.

1.6.5.1 Patrón Experto en Información.

Problema: En un sistema con cientos de clases, no se asignan las responsabilidades a quién creamos deba tenerla por su nombre; sino a quién tenga la mayor cantidad de información para asumir la responsabilidad.

Solución: Asignar una responsabilidad al experto en información: la clase que cuenta con la información necesaria para cumplir con la responsabilidad.

Beneficios del Patrón Experto en Información.

- Se conserva el encapsulamiento.
- El comportamiento se distribuye entre todas las clases implicadas en la ejecución de la función.

1.6.5.2 Patrón Creador.

Problema: ¿Quién debería ser responsable de crear una nueva instancia de una determinada clase?

Solución: Asignarle a la clase B la responsabilidad de crear una instancia de la clase A en uno de los siguientes casos.

- B agrega los objetos A.
- B contiene los objetos A.
- B registra las instancias de los objetos A.
- B tiene los datos de inicialización que serán trasmitidos a A cuando este objeto sea creado (B es un Experto respecto a la creación de A).

Beneficios del Patrón Creador

- Se brinda soporte a un bajo acoplamiento.
- Mejores oportunidades de reutilización.

1.6.5.3 Patrón Bajo Acoplamiento.

Problema: ¿Cómo diseñar sistemas con una dependencia escasa y una alta posibilidad de reutilización?

Solución: asignar una responsabilidad para mantener bajo el nivel de dependencia entre clases.

Beneficios del Patrón Bajo Acoplamiento

- Las clases no se afectan por cambios de otros componentes.
- Fáciles de entender por separado.
- Fáciles de reutilizar.

1.6.5.4 Patrón Alta Cohesión.

Problema: ¿Cómo mantener la complejidad dentro de límites manejables al diseñar las colaboraciones entre clases?

Solución: asignar una responsabilidad garantizando que la cohesión (colaboración, cooperación) entre las clases se mantendrá alta.

Beneficios del Patrón Alta Cohesión

- Mejora la claridad y la facilidad con que se entiende el diseño
- Se simplifican el mantenimiento y las mejoras en funcionalidad.
- A menudo se genera un bajo acoplamiento.
- Mayor capacidad de reutilización.

1.6.5.5 Patrón Controlador.

¿Qué es un Controlador?

Objeto de interfaz no destinada al usuario (interfaz no gráfica) que se encarga de manejar un evento del sistema, definiendo el método a operar. Dicha clase puede representar las siguientes opciones:

- El sistema global (de fachada).
- La empresa u organización global (de fachada).
- Algo en el mundo real que es activo y puede participar en la tarea (de tareas).
- Un manejador artificial de todos los eventos del sistema relacionados con un CU (de CU).

Problema: ¿Quién debería ser responsable de atender un evento del sistema?

Solución: asignar la responsabilidad del manejo de un mensaje de los eventos del sistema a una clase.

Beneficios del Patrón Controlador

- Mayor potencial de componentes reutilizables.
- Rastreo sobre el estado de un caso de uso.

Podemos llegar a la conclusión de que los patrones son estándares o normas que codifican buenas prácticas de diseño. Diseñar cumpliendo patrones garantiza soluciones sencillas y reutilizables.

1.7 Conclusiones.

El diseñador de sistemas debe ser capaz de recibir el análisis y transformar la lista de requisitos del usuario en un diseño arquitectónico de alto nivel que proveerá las especificaciones a los programadores. Además debe existir una total retroalimentación con el análisis desarrollado porque su éxito va a depender de la calidad con la que se haya realizado el análisis.

CAPITULO 2. ROL DISEÑADOR DE SISTEMA.

2.1 Introducción.

Este capítulo está completamente dedicado a explorar, el desarrollo del diseño del framework Mever, el estilo arquitectónico utilizado, patrones de diseño utilizados, y los artefactos del modelo del diseño.

2.2 Mever-framework (Mergedor de Versiones).

Mever es un framework diseñado para poder hacer persistente la información contenida en los diagramas de flujo de información (DFI) que se modelan en CheSys. De una forma transparente para el usuario y con el menor costo posible, en un formato no volátil y volver a recuperar dicha información para que el sistema los vuelva a generar según la información contenida en el fichero. Su nombre se debe al propósito primario intercambiar información con versiones futuras.

Actualmente en la universidad no existe un framework que guarde el estado completo de una familia de objeto en un fichero que sea portable y que tenga en cuenta los requisitos no funcionales. Normalmente el proceso de serializar un objeto no es complicado, pero hay que tener muchas detalles presente, lo que conlleva abrir puertas a la introducción de errores sutiles, que sean difíciles de detectar y corregir.

2.2.1 Patrones utilizados en el diseño del framework.

- **Patrón Layers.**

El patrón Layer [12] es aplicable a muchísimos tipos de sistemas. Este patrón define como organizar el modelo de diseño en capas lo cual quiere decir que cada componente de una capa

solo puede hacer referencia a componentes que estén en las capas inmediatamente inferiores. Este patrón es importante por que simplifica la comprensión y organización del desarrollo de sistemas complejos, reduciendo las dependencias de forma que las capas más bajas no sean conscientes de ningún detalle o interfaz de las superiores. Además no ayuda a identificar que se puede reutilizar.

- **Estilo Arquitectónico MVC.**

La arquitectura del Framework se basa en la arquitectura MVC [17] (patrón de diseño), donde existe un proceso de abstracción que permite dividir la aplicación entre los componentes involucrados en mantener los datos y los que presentan los datos.

De esta forma se minimiza la dependencia entre ellos y permite que diferentes desarrolladores o equipos de desarrolladores puedan cumplir perfiles diferenciados. Lo más importante de esta arquitectura es que permite que futuros cambios técnicos o de aparición de estándares tengan el mínimo impacto sobre lo ya desarrollado.

La arquitectura MVC divide los objetos o componentes implicados en una aplicación en 3 tipos:

- Modelo: Representa los datos de la aplicación y las reglas de negocio, que en algunos casos gobiernan el acceso y la modificación a estos datos. El modelo, notifica a las vistas de sus cambios y proporciona todos los métodos necesarios para que las vistas puedan conocer la información que contienen.

- Vista: Muestra el contenido del modelo. Accede a los datos del modelo y especifica de qué manera se muestran estos datos. La Vista también se encarga de redireccionar todos los eventos / acciones del usuario hacia él.

- Controlador: Define el comportamiento de la aplicación, interpreta los eventos del usuario y mapea las acciones a realizar sobre el modelo. Estas acciones pueden ser realizar un cambio al estado del modelo o bien ejecutar funciones de la lógica de negocio del modelo. Dependiendo de la acción realizada por el usuario, el Controlador puede seleccionar una nueva vista a mostrar en respuesta a la petición realizada por el usuario.

En todos los subsistemas están presentes los cinco patrones, de asignación de responsabilidades más básicos (GRASP), Experto, Creador, Bajo Acoplamiento, Alta Cohesión, Controlador y también algunos patrones de diseño GOF.

- **Solitario / Singlenton.**

Se necesita una instancia única de las clases fachada y ShmFactory, para poder tener puntos de acceso global a las mismas, y tener el acceso más controlado de sus responsabilidades, ya que fachada es contenedora de todos los subsistemas del framework y ShmFactory es la fábrica de objetos para guardar un archivo de forma personalizada.

- **Fachada / Facade.**

Como lo dice el nombre es la viva implementación del patrón Facade del grupo de los cuatro (GOF). Se implementó con el objetivo de proporcionar una interfaz simple, para dar soporte a todos los subsistemas complejos de Mever-framework, todas sus dependencias y reducir el número de objetos con los que la vista trata. La “Fachada” sabe qué clase es responsable de cada petición, pero es completamente transparente para ellas. Por tanto, aunque las funciones las desempeñan las otras clases, es a la “Fachada” a quien le toca el trabajo de “traducir”.

- **Método de fábrica / Factory Method.**

Se construyó una clase ShmFactory para la creación de varios objetos, y de esta forma la clase pueda adelantar, las clases de objetos que debe crear. Se ganó en flexibilidad, pues crear los objetos dentro de una clase con varios “Método de Fábrica” es siempre más flexible que hacerlo

directamente, debido a que se elimina la necesidad de atar nuestra aplicación a unas clases de productos concretos

- **Acción / Command.**

Con este patrón al encapsular en un objeto la acción que satisface una petición, el usuario no tiene que preocupase por las tareas que se tiene que realizar antes de lograr salvar o cargar un fichero, solamente en la acción de debe de hacer, ya que con este patrón diferentes objetos pueden ejecutar la misma acción sin necesidad de repetir su declaración e implementación, Se puede cambiar dinámicamente con facilidad, la acción que realiza o está asociada a un objeto. En este caso la clase MeverCommand encapsula las acciones de leer y escribir un fichero.

- **Estrategia / Strategy.**

Mever-*framework* tiene una familia de algoritmos, que tienen el mismo propósito, leer y escribir los archivos que se generan en el proyecto. Cada uno de estos algoritmos está encapsulado por separados. Cada clase representa una estrategia, para ejercer la lectura y escritura.

El encapsulamiento de algoritmos en clases separadas, ofrece una ventajosa alternativa a la especialización por herencia del contexto para obtener un comportamiento diferente, que promueve la independencia, la facilidad de entender el diseño y la posibilidad de futuras extensiones.

2.2.2 Microsoft UIPAB.

El diseño del UIPAB hace de él un componente muy flexible. Por sí solo ya cumple una función, pero además se puede extender todo lo que se quiera de una manera muy fácil.

Con lo de que por sí solo ya cumple las siguientes tareas:

- **Abstraer el flujo de navegación de las interfaces:**

El flujo de navegación de una aplicación es una regla de negocio y, como tal, no debería estar implementado en la capa de presentación. Más adelante se observará que realmente lo que está haciendo es sacar el flujo de navegación del código en sí de la aplicación, lo que aumenta la flexibilidad y portabilidad de ésta.

- **Facilitarnos la reutilización del modelo de una aplicación en otra**:

Muchas veces se está desarrollando una aplicación de escritorio que probablemente más adelante necesite tener un interfaz Web, y también sucede en orden inverso. Mediante la utilización del UIPAB el modificar el interfaz de la aplicación se vuelve una tarea realmente sencilla.

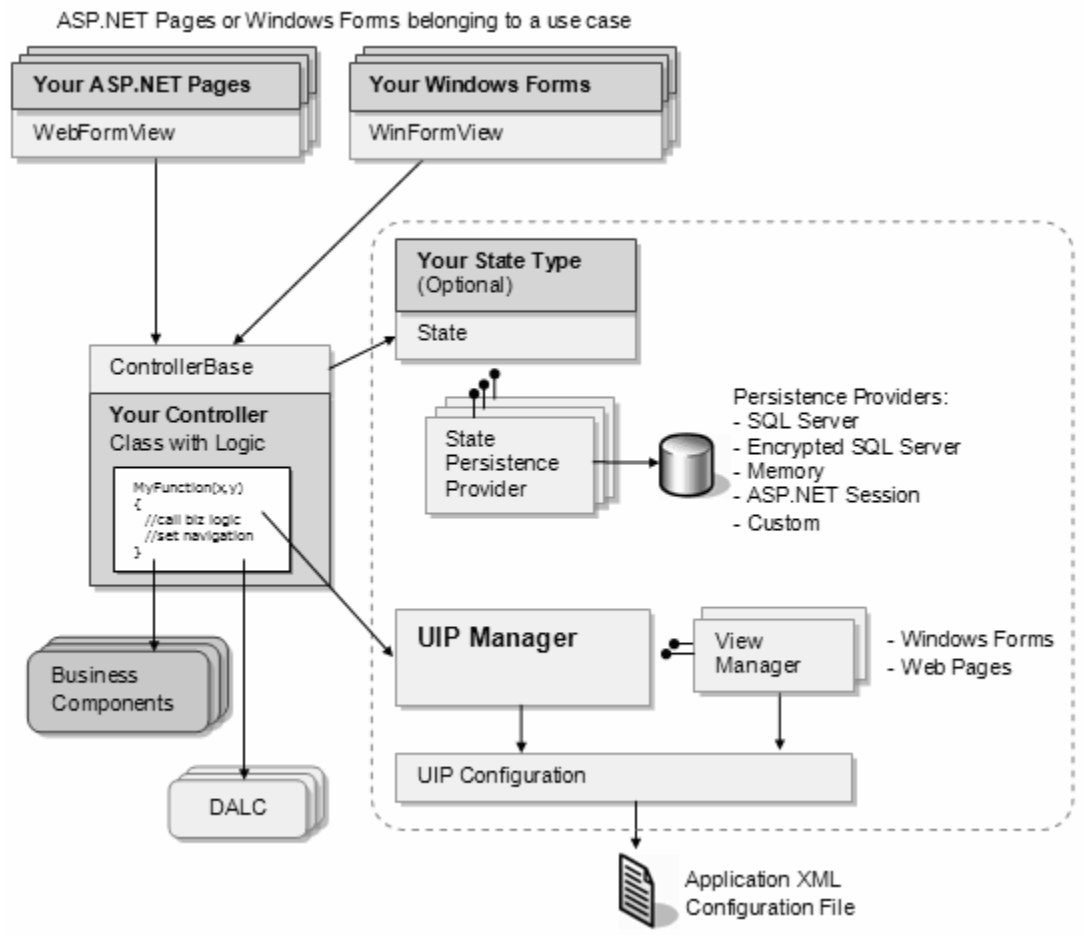


Ilustración 7. Arquitectura de UIPAB.

- **Ofrecer un sistema de persistencia entre sesiones para el estado de la aplicación:**

Es algo común también el tener grupos de operaciones que, o se realizan en conjunto, o no se debe realizar ninguna de ellas. Si se imagina por ejemplo una aplicación para subir un punto en la nota de todos los estudiantes en una asignatura. La aplicación empezaría subiendo un punto al estudiante A, luego otro a B, luego otro a C y se cae la conexión. Sucede que el estudiantes C se quedó sin un punto, mientras que A y B ya tienen un punto más en su nota. Esto se suele hacer con lo que se denominan transacciones, de forma que si no se completa el proceso, se deshace automáticamente lo que se había hecho hasta el momento.

Es decir, se marca la tarea como atómica, se hace entera o no se hace nada. Pero suponiendo que se tiene una transacción en una aplicación Web en la que el usuario tiene que ir visitando varias vistas de la aplicación y haciendo tareas… y de repente se reinicia el ordenador o se confunde y abre otra Web… sería muy pesado hacerle empezar de nuevo.
Pues para esto el UIPAB proporciona un mecanismo de persistencia para la sesión del usuario, de forma que éste pueda retomar ese proceso donde lo dejó, sin tener que volver a empezar (pero sin dejar el sistema en un estado inconsistente, claro).

2.2.2.1 Capa de presentación con el UIPAB.

El UIPAB [14] se basa en la aplicación del patrón MVC a la capa de presentación de nuestra aplicación.

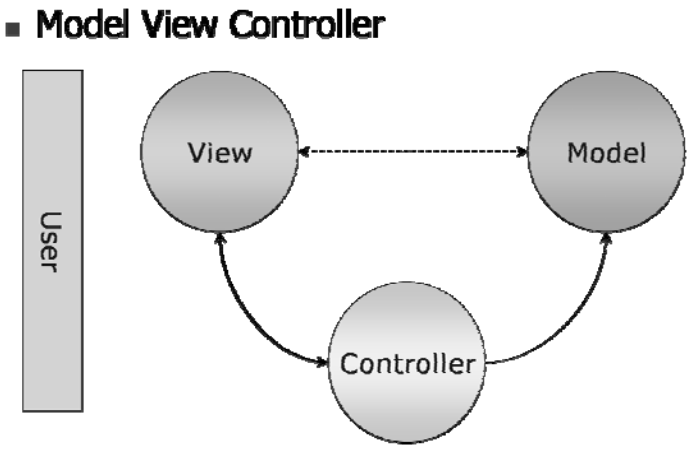


Ilustración 8. Patrón MVC.

El UIPAB nos divide la interfaz en dos capas diferentes: los User Interface Componentes y los User Interface Process Componentes. Los primeros son los controles, formas con los que el usuario interactúa, mientras que los segundos son los que gestionan esa interacción, es decir, establecen la navegación, acceden al “modelo” para obtener información, etc. Así, la “vista” del

MVC serían los user interface componentes y el “controlador” los user interface process componentes.

Resumiendo, las principales ventajas del UIPAB son que saca de la interfaz la gestión del flujo de navegación, nos abstrae de la gestión del estado de la sesión de usuario, nos facilita enormemente el pasar de una aplicación Web a una de escritorio (o viceversa) o a una interfaz para dispositivo móvil reutilizando por completo el modelo de la aplicación y por último nos ofrece un mecanismo de persistencia para las sesiones de usuario.

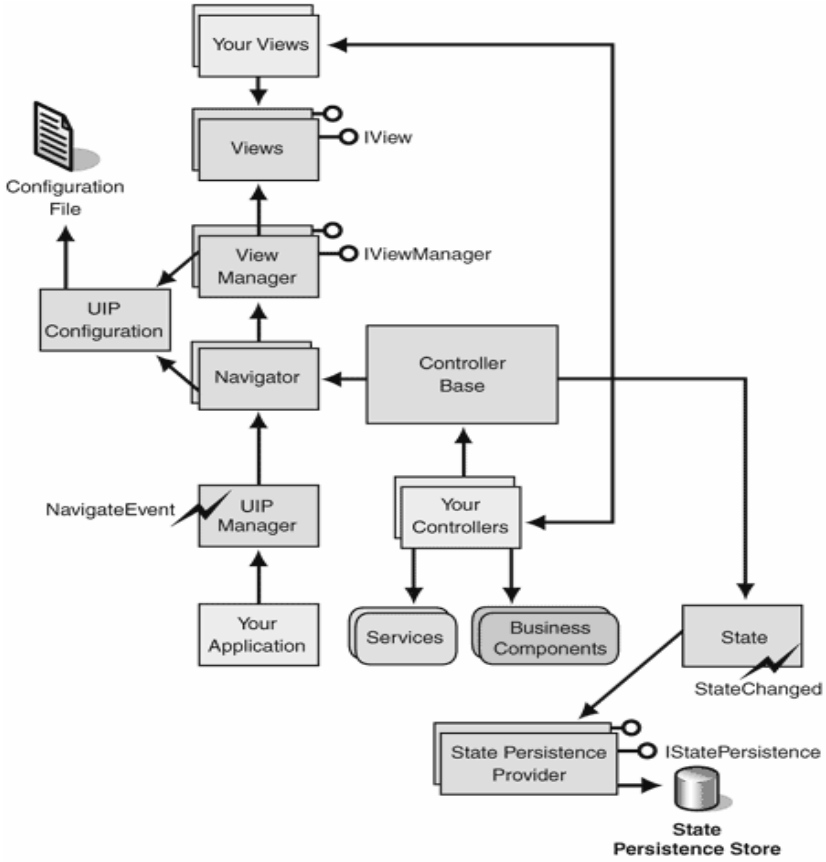


Ilustración 9. Flujo de navegación de UIPAB.

- El flujo de navegación de la aplicación, está en un fichero externo en XML. Así, se puede en cualquier momento cambiar ese flujo de navegación, añadir vistas. Básicamente se

basa en la idea de tener una serie de “vistas” sueltas en la aplicación, y utilizar ese fichero XML para “coserlas” (vamos, creando un grafo no dirigido con las vistas como nodos). Con esto además conseguimos una gran adaptabilidad, es decir, lo típico de que si un usuario es administrador y hace tal cosa pues que vaya a este sitio, que si no es administrador vaya a tal otro sitio, que si pertenece a este departamento pase antes por esta otra vista, etc. Y si queremos cambiar estos mapas de navegación, no tendremos que cambiar una sola línea del código de nuestra aplicación (con las grandes ventajas que ello conlleva), únicamente nos servirá con modificar ese fichero XML, que no deja de ser texto plano.

- Abstrae de la gestión del estado de la sesión de un usuario en el sentido en que tiene un componente específico para esta tarea, independiente de la interfaz de usuario en sí. Así, las vistas saben únicamente dónde encontrar la información sobre el estado, pero no tienen ni idea sobre cómo nosotros guardamos esa información, dónde la guardamos o cómo esa información afecta a otras vistas.

- Gracias a este último punto, entre otras cosas, facilita enormemente el paso de una aplicación Web a otra de escritorio, por ejemplo. Al abstraernos de la gestión de la sesión de usuario a una clase que se encargue de ello, podemos cambiar fácilmente de una interfaz a otra, y hacer que ésta última obtenga la información sobre la sesión de la misma clase que utilizábamos con la otra interfaz. Además, como las vistas en sí no conocen nada en absoluto sobre el flujo de navegación o su relación con otras vistas, hemos conseguido que las vistas sean completamente independientes del resto de la aplicación. Con cambiar las vistas hemos cambiado por completo la interfaz de la aplicación, manteniendo los mismos controladores y la misma clase gestora del estado (o sea, mantenemos el “modelo” y el “controlador”, modificando únicamente la vista)
- Por último, al independizar la gestión de la sesión también facilita las labores de persistencia de ésta. Concretamente nos proporciona un mecanismo automático para ello.

2.2.3 Propuesta del Framework.

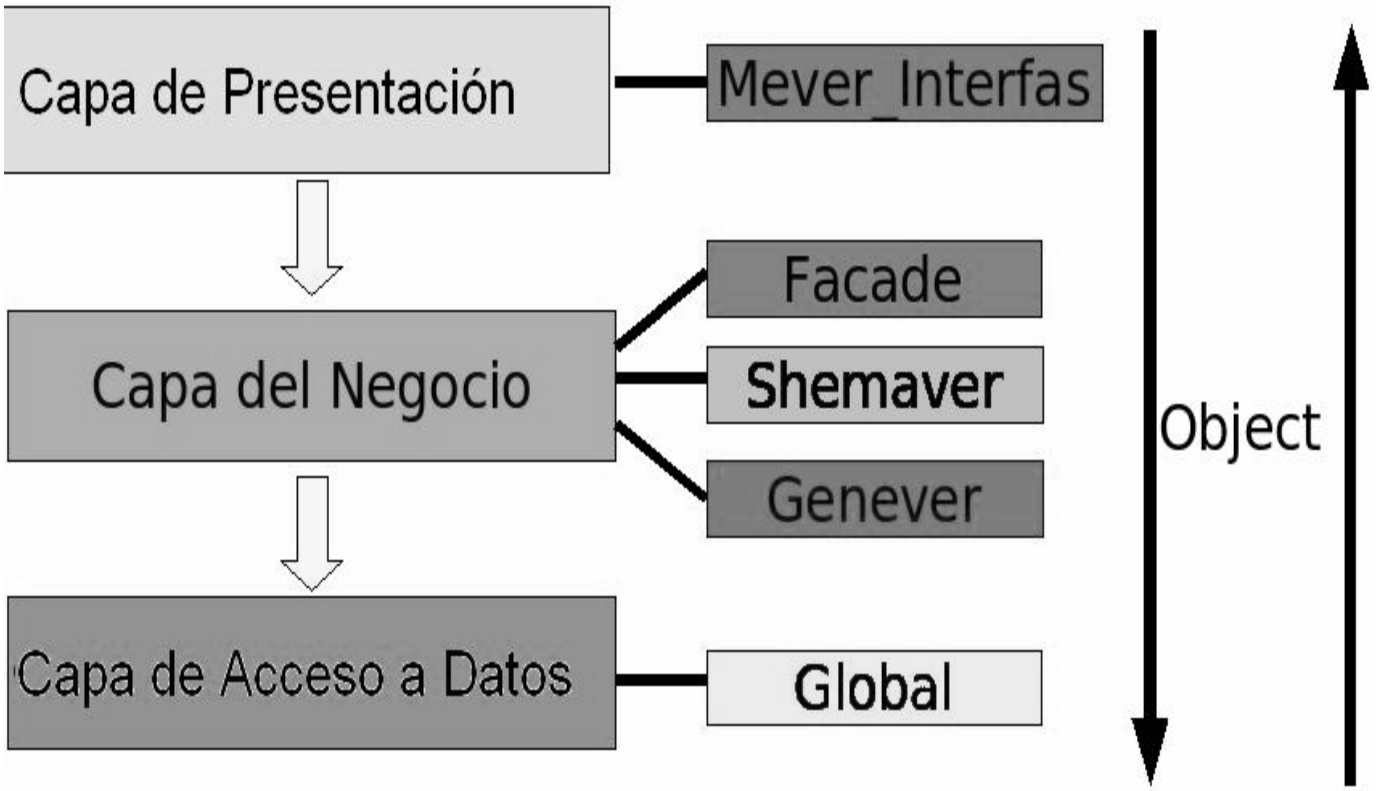


Ilustración 10. Arquitectura Mever-Framework.

Mever-*framework* tiene una arquitectura multicapa, con diferentes módulos que cada uno lleva a cabo una tarea específica.

- Mever-Interfaz, compuesta por varios formularios donde cada uno se especializa en realizar una tarea específica, y también el formulario principal que da la opción de trabajar en una tarea específica.

- Facade, conoce que subsistemas son responsables de que peticiones y así, delega las peticiones a los componentes apropiados. Simplifica el acceso a un conjunto de objetos relacionados, a través de un objeto que todos los otros objetos fuera del conjunto, usan para comunicarse con los objetos del conjunto.

- Subsistema “Shemaver”. Esquema para control de versiones *(Shemaver),* se encarga de preparar un esquema que representa una plantilla para la serialización personalizada en un fichero XML con una estructura interna que nos permite conocer toda la información concreta del proyecto que realmente se necesita salvar. El motor del framework Mever para la serialización profunda utiliza una técnica parecida de esquema pero no con un fichero XML sino con una clase que esta especializada en esa tarea con un mecanismo de reflexión. Su nombre se debe precisamente a esa funcionalidad.

- Subsistema “Adaver”. Adaptador de versiones *(Adaver),* está especializado en adaptar los ficheros viejos en los de las futuras versiones y de esta forma recuperar información y no tener que modelar otra vez los DFI. Este módulo es muy necesario para los proyectos que tengan incompatibilidad de versiones con las salvas de información.

- Subsistema “Genever”. Genera el fichero Marco_File.cs para la serialización y deserialización. Este módulo por solo generar un fichero de deja de tener importancia ya que es el contenedor de todas las clases del proyecto que se necesitan para realizar el proceso de salvar y/o recuperar la información. En realidad lo que el motor de Mever necesita para poder llevar a cabo los procesos anteriormente mencionados es un arreglo

de objetos (type) , esto lo podemos conseguir a través de los mecanismos de reflexión que ofrece .net framework, pero la información recogida no es útil para la serialización profunda aunque aparentemente sean el mismo tipo de objeto, la única solución es obtener el mismo arreglo pero editarlo de forma manual con el operador (typeof()), pero claro que editar el arreglo no es algo cómodo , ni rápido cuando se tiene un gran proyecto ; el simulador CheSys es el ejemplo práctico para emplear esta aplicación ya que tiene más de trescientos objetos de tipo Type que hacen referencia a clases e interfaces.

- Subsistema "Global" es la máquina de serialización y deserialización de Mever–*framework.* Subsistema que contiene todas las clases que se especializan en el proceso de salvar y/o recuperar la información del proyecto. Este subsistema está diseñado para poder trabajar con cualquier mecanismo de .net framework de serialización y/o deserialización.

2.2.4 Almacenamiento persistente en Mever-Framework.

2.2.4.1 Serialización profunda y personalizada.

Cuando se desea guardar la información de forma profunda y personalizada lo primero que se hace es generar el un esquema en formato XML, con lo que realmente se desea guardar en toda la aplicación utilizando el subsistema *Shemaver* este proceso es controlado y guiado por controla el usuario paso a paso de una forma muy sencilla a través de un wizard, donde el usuario tiene el control de todas las clases del proyecto que son persistente y va seleccionando las que realmente le interesan y toma de ellas los campos y propiedades toda esta información al final queda recogida y se guarda como una plantilla o esquema y por esta platilla siempre se guiará el motor para de esta forma guardar la información.

Esta plantilla la podrá revisar el usuario claramente después haberla creada y guardada porque queda en un formato totalmente claro y transparente para el usuario el único requisito es que

tenga un conocimiento del lenguaje XML. Después el motor deserializa la platilla, explora cada clase que el usuario había seleccionado anteriormente en el wizard con su información correspondiente, y solo guarda lo que hay en ella. Guardar esta información en un fichero tiene muchas ventajas, la primera es que evita que el proceso sea repetitivo cada vez que se necesite guardar la información de esta manera, la segunda es que en proyectos pequeños este fichero se puede editar manualmente y en proyectos grandes también se puede pero no se recomienda debido a la complejidad que puede tener este fichero y se pueden cometer sutiles errores y difíciles de encontrar.

Ejemplo aquí

```
1. Type tmp = typeof (Test.Estudiante);
2. Facade.Facade myfacade = Facade.Facade.getIntance();
3. myfacade.Accion = new Global.ShmHelper();
4. myfacade.ShmCreateClassInfo(tmp.GetType());
5. myfacade.ShmCreateMenver("edad");
6. myfacade.ShmCreateMenver("sexo");
7. myfacade.ShmCreateMenver("Nombre");
8. myfacade.Serilize(myfacade.Template,"Template.xml");
```

```
namespace Test
{
        /// <summary>
        /// Descripción breve de Persona.
        /// </summary>
        ///
        public enum Sexo
        {
                Masculino,Femenino
        }
        public class Estudiante
        {
                private string nombre;
                private byte edad;
```

```
                private char[] ci;
                private Sexo sexo;

                public Estudiante()
                {
                        this.ci = new char[11];
                        this.ci = "83060425300".ToCharArray();
                        this.edad = 23;
                        this.sexo = Sexo.Masculino;
                        this.nombre = "Enaide C Fernandez";
                }



public char[] CI
                {
                        get{return this.ci;}
                        set{this.ci = value;}
                }
                public string Nombre
                {
                        get{return this.nombre;}
                        set{this.nombre = value;}
                }
                public Sexo Sexo
                {
                        get{return this.sexo;}
                        set{this.sexo = value;}
                }
        }
}
```

Ejemplo del esquema a partir de la clase

```
 <?xml version="1.0" encoding="utf-8" ?>
<Template xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   - <ClassInfo>
```

```
        <Name>Estudiante</Name>
        <FieldInfo TypeInfo="Byte" NameInfo="edad" FormatInfo="Byte edad" />
        <FieldInfo TypeInfo=" Test.Sexo" NameInfo="sexo" FormatInfo=" Test.Sexo sexo" />
         <PropertyInfo TypeInfo="System.String" NameInfo="Nombre"
FormatInfo="System.String Nombre" />
- </ClassInfo>
</Template>
```

En la primera línea se crea una objeto Type para representar una instancia de la clase Estudiante, en la segunda creamos un objeto de Facade y después en la tercera se especifica el comando concreto o la acción que debe hacer, en este caso se debe realizar una serialización poco profunda, después en la cuarta línea se crea un objeto ShmClassInfo con el objeto Type que se había creado en el primer paso, la clase ShmClassInfo es la contenedora de todo lo que se quiera guardar de la clase Estudiante, ya en los pasos cinco ,seis y siete para ir terminando se selecciona lo que se quiere y finalmente se guarda esta información en el paso número ocho.

Como se puede observar en fichero se registró que siempre de la clase Estudiante solamente se guardarán los campos privados (*FieldInfo)* sexo y edad y las property *(PropertyInfo)* Nombre. El framework de .net tiene mecanismos para hacer esto mismo con el atributo [NoSerialized], el único requisito es darle permiso a la clase para que se pueda serializar, esto se consigue con [Serializable]. También implementando la interfaz ISerializable, se puede obtener un resultado parecido. Esto es muy útil en caso que se tenga información importante y no se quiera guardar, ejemplo contraseñas o claves de encriptación como se pone ejemplo siguiente.

```
[Serializable]
public class MyObject
{
        [NoSerialized]
        public string _MySuperSecretPassword;
        // …
```

```
    }
```

El primer ejemplo anterior tiene el inconveniente que una vez marcada la clase como [Serializable], la clase no tendrá la posibilidad de tener control de versiones. Esta opción sólo debe utilizarse cuando se esperen cambios mínimos entre las diferentes versiones de una clase. El segundo mecanismo es el más conveniente pero hay que tener ciertos requisitos como darle permiso para serializarla con el atributo [Serializable], implementar la interfaz ISerializable y también adicionarle un constructor especial para poder leer a salva del fichero, en este caso ya no es necesario [NoSerialized] porque esta interfaz tiene el método GetObjectData para especificar la información importante de la clase.

```
[Serializable]
public class MyObject : ISerializable
{
public int n1;
public int n2;
public String str;

public MyObject()
{
}

protected MyObject(SerializationInfo info, StreamingContext context)
{
n1 = info.GetInt32("i");
n2 = info.GetInt32("j");
str = info.GetString("k");
}

public virtual void GetObjectData(SerializationInfo info, StreamingContext context)
{
info.AddValue("i", n1);
info.AddValue("j", n2);
info.AddValue("k", str);
}
}
```

Sin duda todo este trabajo se ahorra con el motor de serialización que tiene Mever-*framework* y además el código queda mucho más limpio ya que el programador no tiene que centrarse en programar todos estos detalles ni tener en cuenta si una clase en un futuro tendrá diferentes versiones y no tener que inyectar código en la clase o adicionarle responsabilidades que no tengan que ver con su propósito.

El motor de serialización realiza un seguimiento de todas las clases y el contenido guardado en ellas en el fichero XML y solo se centra en serializar su contenido. El único requisito que deben cumplir los gráficos de objeto es que todos los objetos mencionados por el objeto que está siendo serializado deben estar registrados en fichero MarcoFile.cs. de lo contrario, se enviará una excepción cuando el motor intente serializar un objeto no marcado.

Cuando se desea recupera la información de un fichero, el motor no interpreta la información contenida en el archivo Template.XML, porque este pudo haber sido modificado y tener algún error, tal como pérdida de información, existente en la clase o información adicional y entonces se lanzaría una excepción. Para evitar esto Mever-*framework* tiene una clase llamada CrackSerilizationInfo programada para pedirle al objeto SerilizationInfo de .net framework toda la información asociada al objeto que se está deserializando para realizar este proceso de la mejor forma posible.

La información contenida en la clase SerilizationInfo no es pública para el usuario por lo que hay que acceder a ella a través del mecanismo de reflexión y deserializar a partir de su contenido porque es el que realmente ya .net framework recuperó sin ninguna excepción. La clase encargada de este tipo de salva de información es ControlHelper.

2.2.4.2 Serialización profunda sin esquema o plantilla.

Este tipo de serialización es más sencilla que la anterior por que no hay que generar ningún esquema o plantilla como el tipo anterior solamente cambiar el comando o acción:

```
Facade.Facade myfacade = Facade.Facade.getIntance();
myfacade.Accion = new Global.DefaultHelper();
```

En este caso, el proceso de serialización guarda todo lo referente a la clase ya que accede a través de reflexión para explorar la clase entera. Los requisitos son tener todos los objetos registradores en el archivo MarcoFile.cs.

Mever-*framework* cuenta con dos clases más para salvar clases, NetHelper y ShmHelper, y salvan la información como lo hace .net framework, porque no hacen más que utilizar lo que .net ya tiene implementado. La idea es que este marco de trabajo contenedor de todos los mecanismos para la salvar información, y que se le puedan agregar más.

2.3 Artefactos.

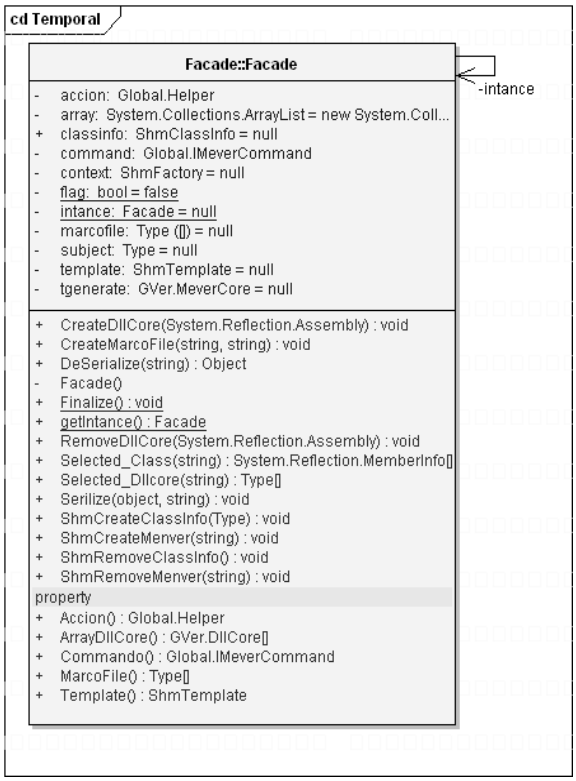
2.3.1 Modelo del Diseño.

Caso de uso: **Salvar/Cargar**

Clases del diseño

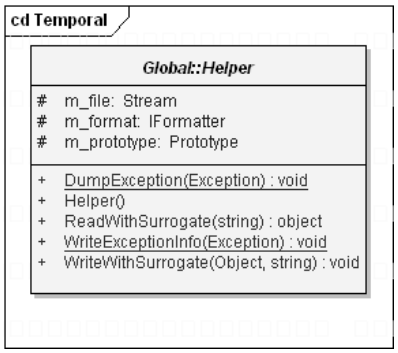
- **Facade**

Es la clase que conoce que subsistemas son responsables de que peticiones y así, delega las peticiones a los componentes apropiados, esta clase es una representación del patrón singlenton.



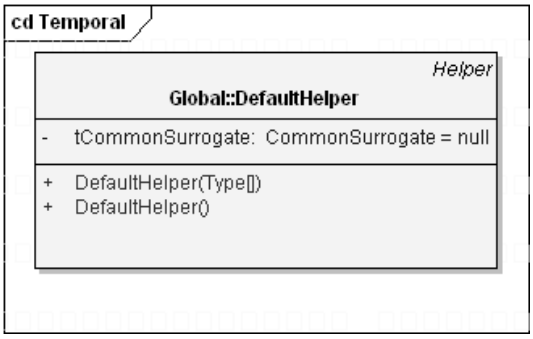
- **Helper**

Es la clase abstracta de las estrategias para salvar y/o cargar un archivo, delega a sus subclases dicha tarea. Representa el patrón estrategia / strategy.



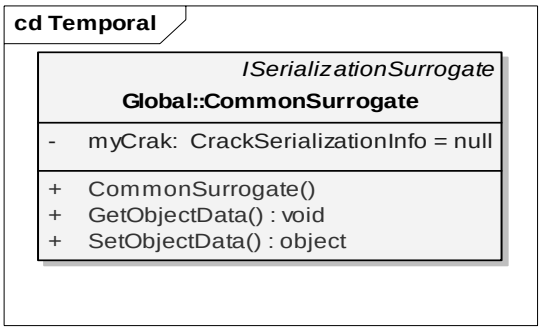
- **DefaultHelper**

Hereda de Helper para definir la estrategia de salvar y/o cargar un fichero con control de versiones.



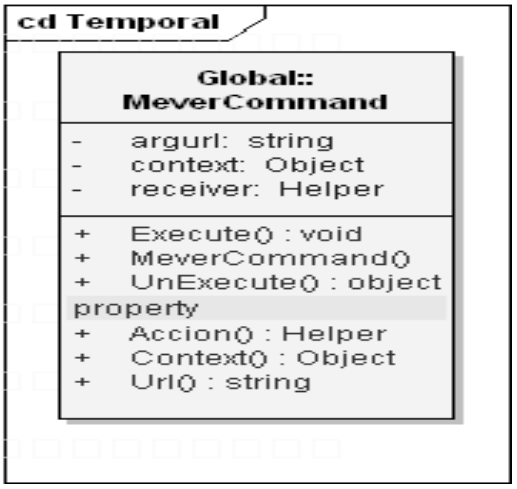
- **CommonSurrogate**

Es la clase que se encarga de dar el formato, para una salva y/o carga los archivos con control de versiones.



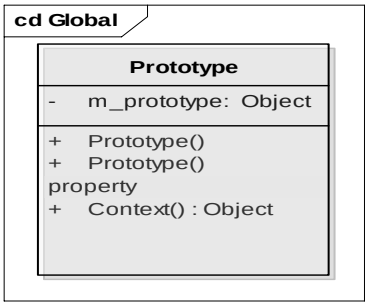
- **MeverCommand**

Esta clase encapsula los comandos de salvar y cargar un fichero, representa el patrón Comando.



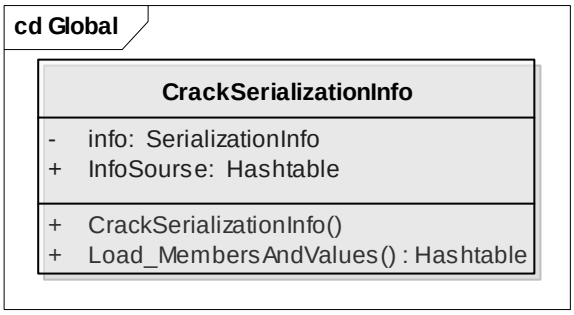
- **Prototype**

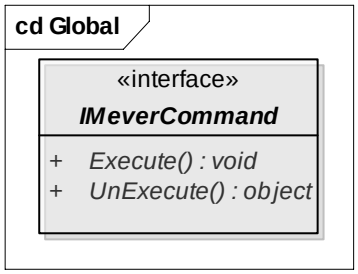
Es la clase entidad donde se almacena y guarda todos los objetos cuya información se desea salvar.



- **CrackSerializationInfo**

Es la clase principal para el proceso de cargar un fichero serializado. Explora la clase deserializada para asignarle dinámicamente los valores los campos que verdaderamente existen en la clase y evitar cualquier tipo de error, en el momento de cargar un fichero.

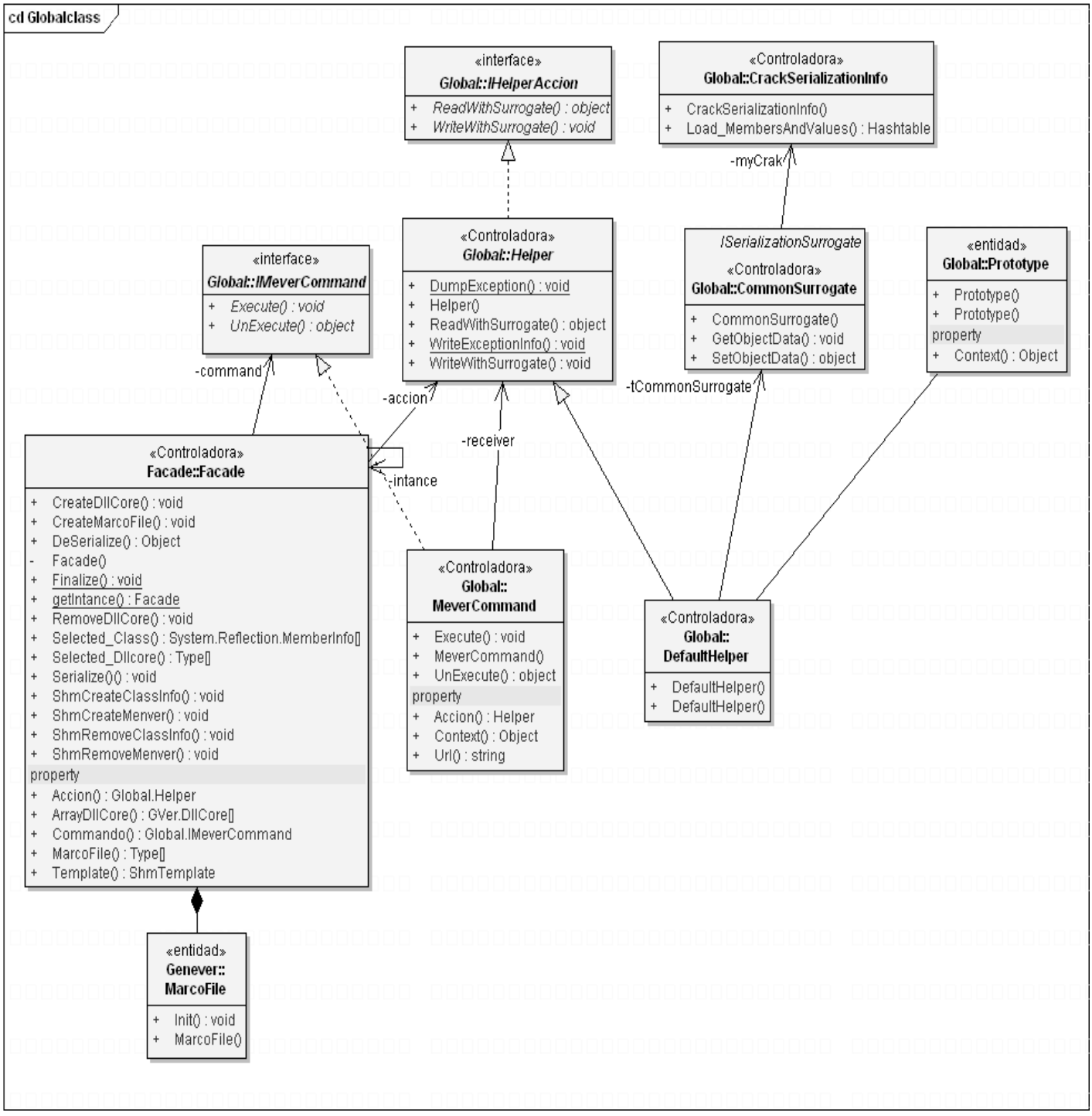




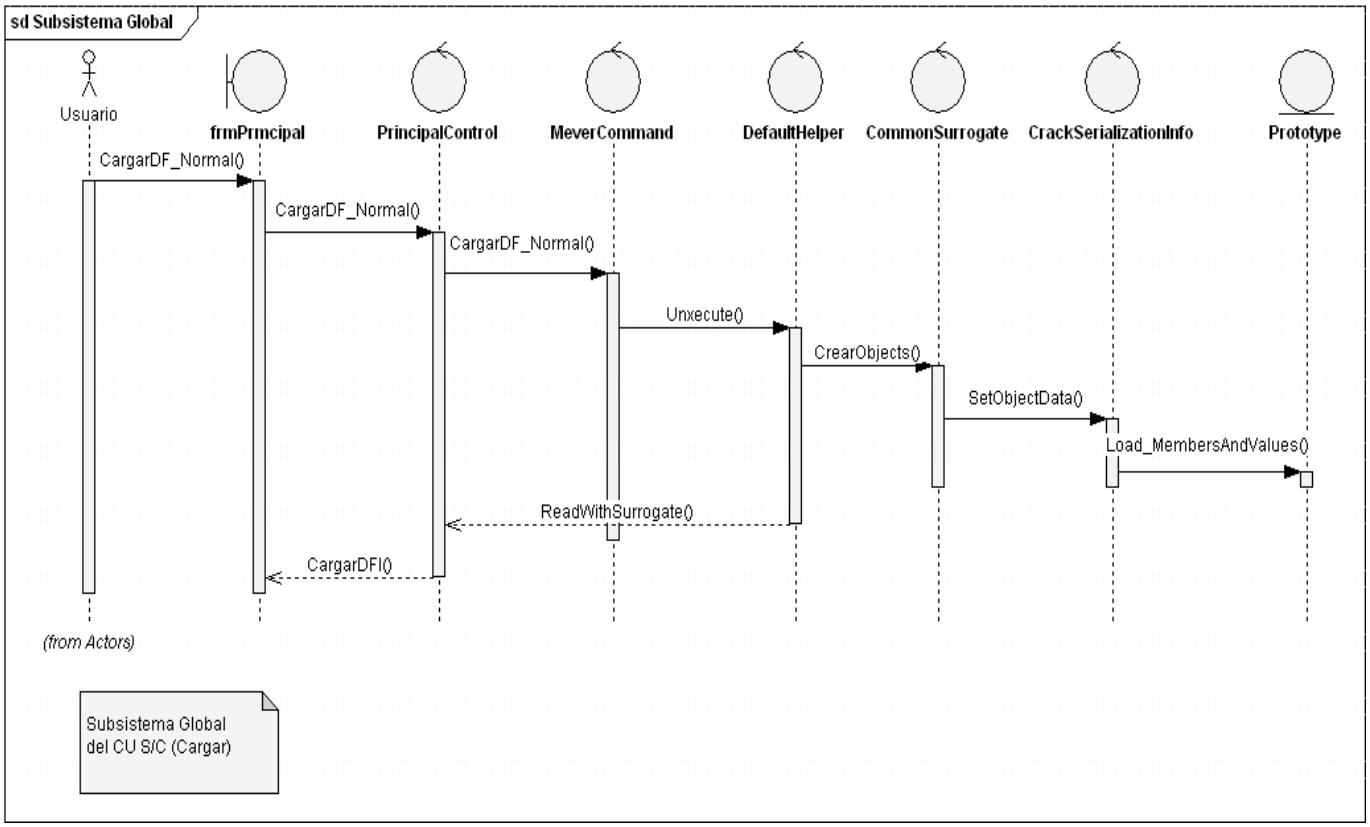
- **IMeverCommand**

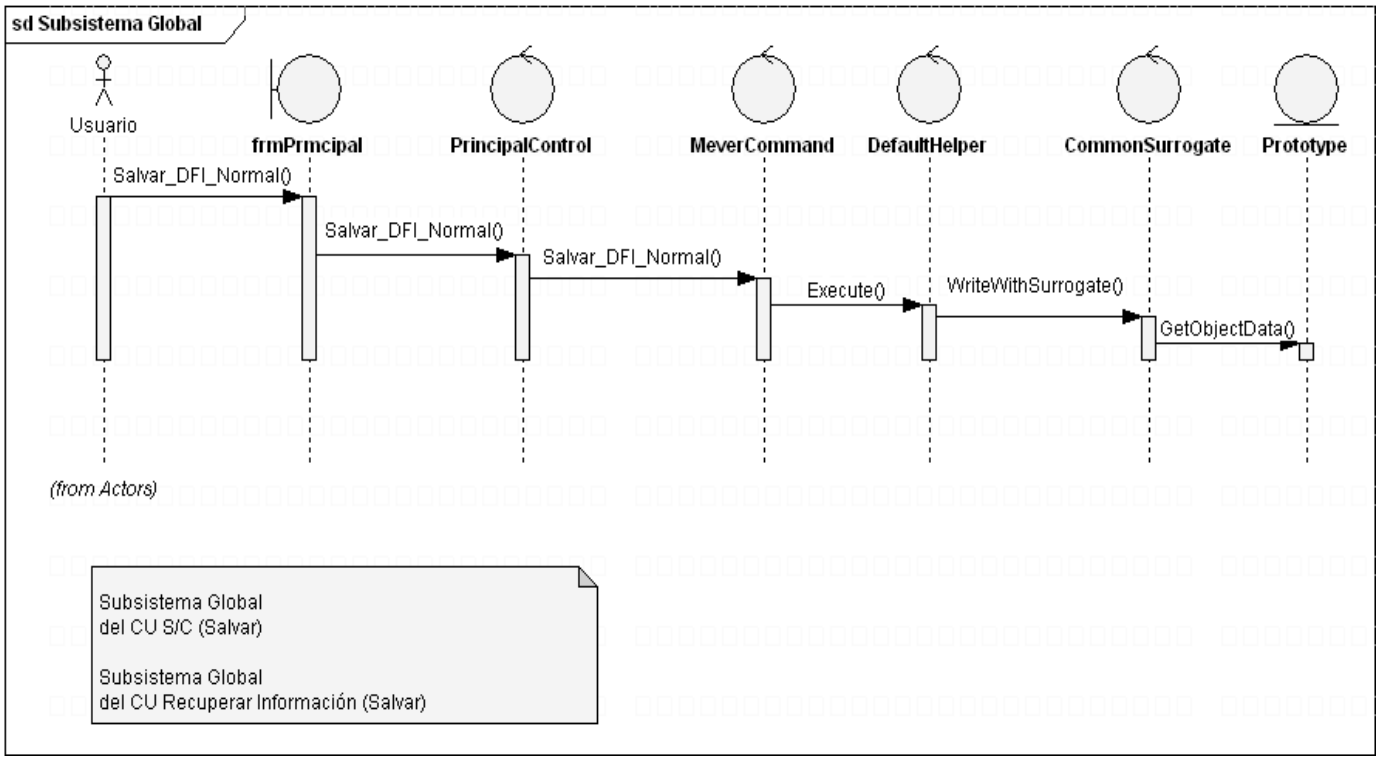
Es la Interfaz que encapsula las operaciones de hacer y deshacer una acción.

Diagrama de Clases



Diagramas de Secuencias



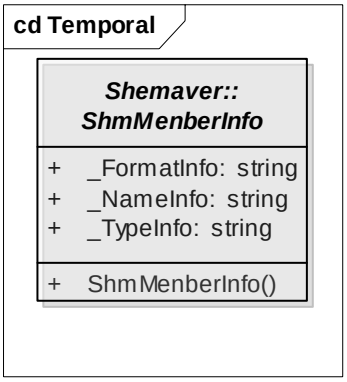

sd Subsistema Global
Usuario
frmPrmcipal
PrincipalControl
MeverCommand
DefaultHelper
CommonSurrogate
Prototype
Salvar_DFI_Normal()
Salvar_DFI_Normal()
Salvar_DFI_Normal()
Execute()
WriteWithSurrogate()
GetObjectData()
(from Actors)
Subsistema Global
del CU S/C (Salvar)
Subsistema Global
del CU Recuperar Información (Salvar)

Caso de uso: **Personalizar Salvar/Cargar**

Clases del diseño

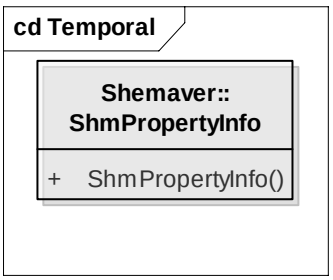
- **ShmMenverInfo**

Es una clase abstracta que representa todos los objetos MenverInfo que explora reflexión y nos interesa hacer persistente su información.



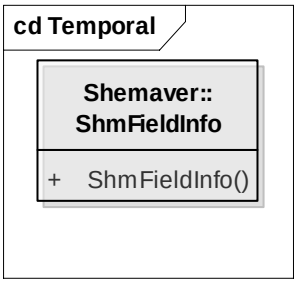
- **ShmPropertyInfo**

Es la clase que encapsula las propiedades/Propertys de los objetos.



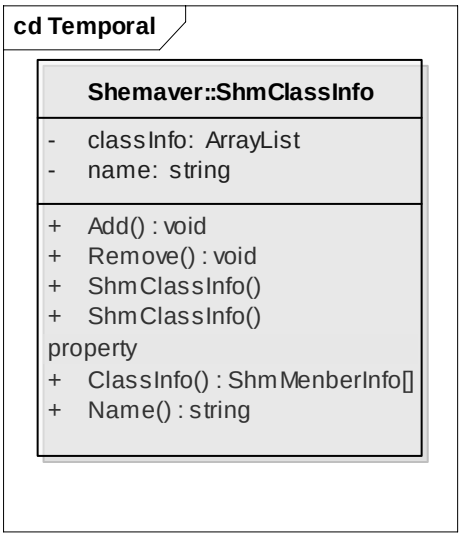
- **ShmFieldInfo**

Es la clase que encapsula los campos/Fields de los objetos.



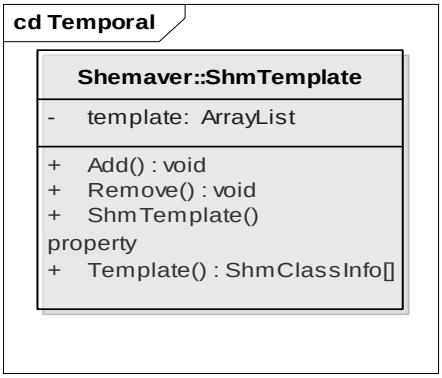
- **ShmClassInfo**

Es la clase que se utiliza para representar las clases que se desean serializar. Esta clase también es contenedora de los objetos shmMenverInfo y shmPropertyInfo.



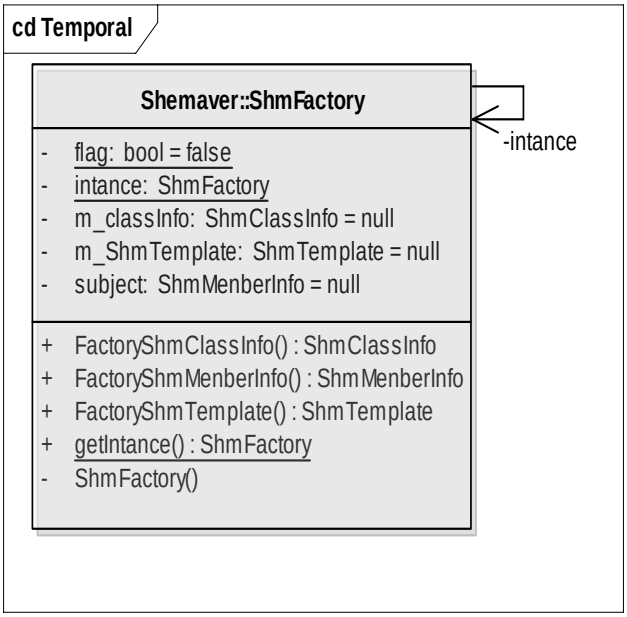
- **ShmTemplate**

Es la clase contenedora de todas las clases que se desean salvar del proyecto.



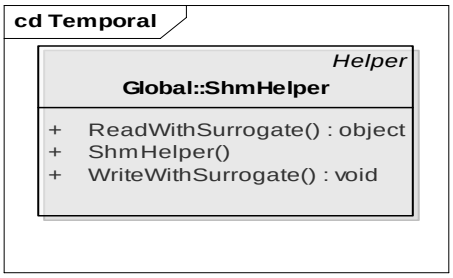
- **ShmFactory**

Es la clase fundamental del subsistema, es la fábrica de objetos de este subsistema.



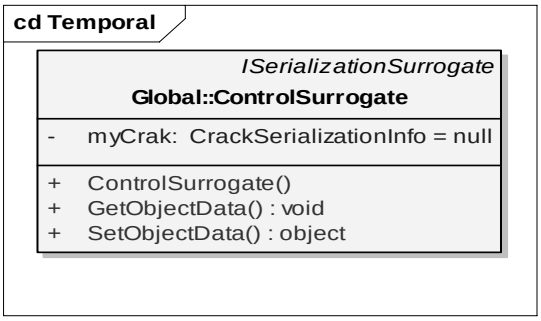
- **ShmHelper**

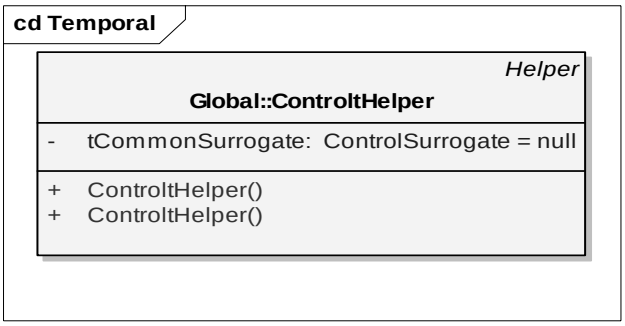
Hereda de la clase Helper para definir la estrategia de salvar y/o cargar un fichero en formato XML estándar.



- **ControlHelper**

Hereda de Helper para definir la estrategia de salvar y/o cargar un fichero con control de versiones, pero de forma personalizada.

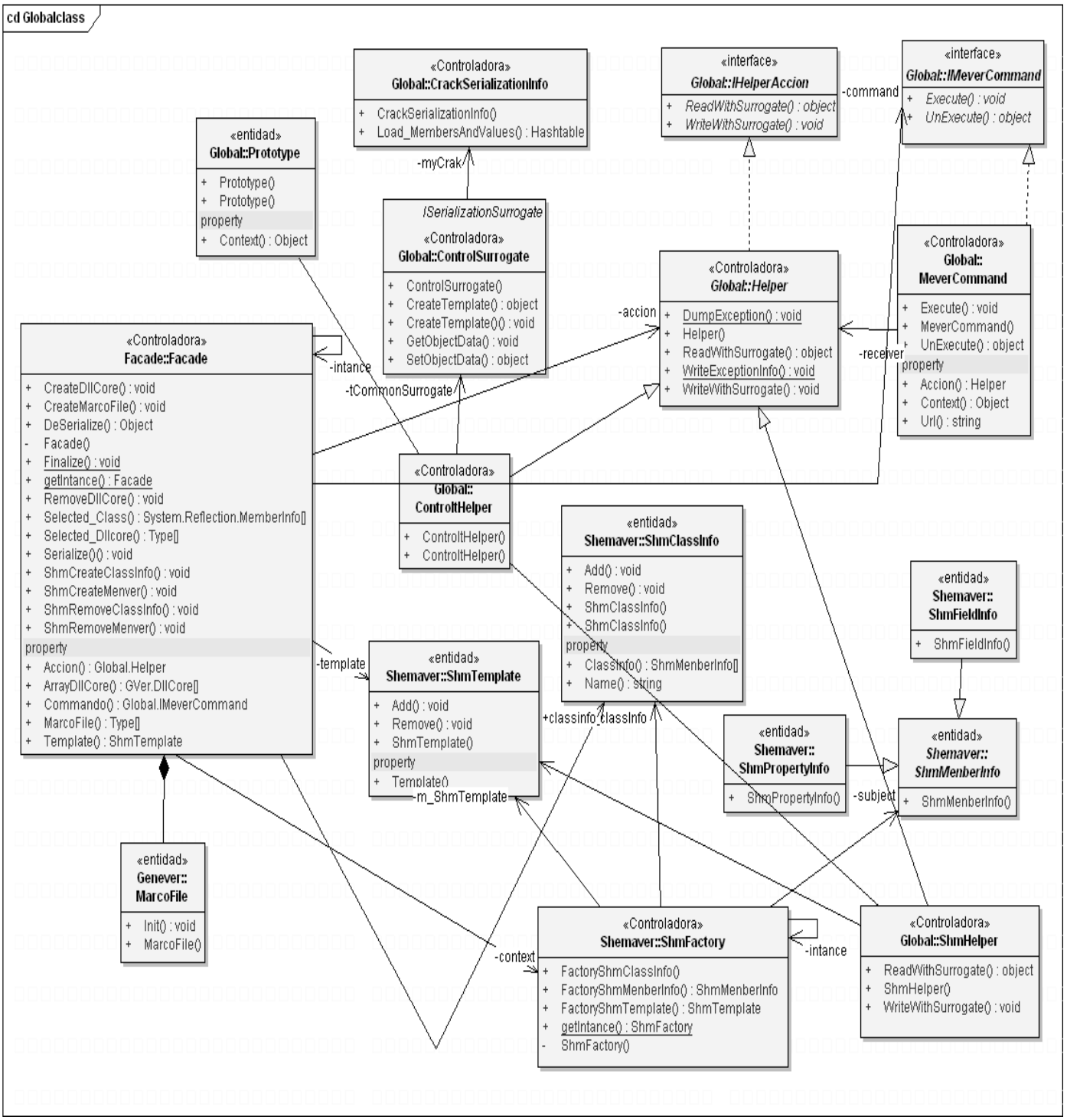




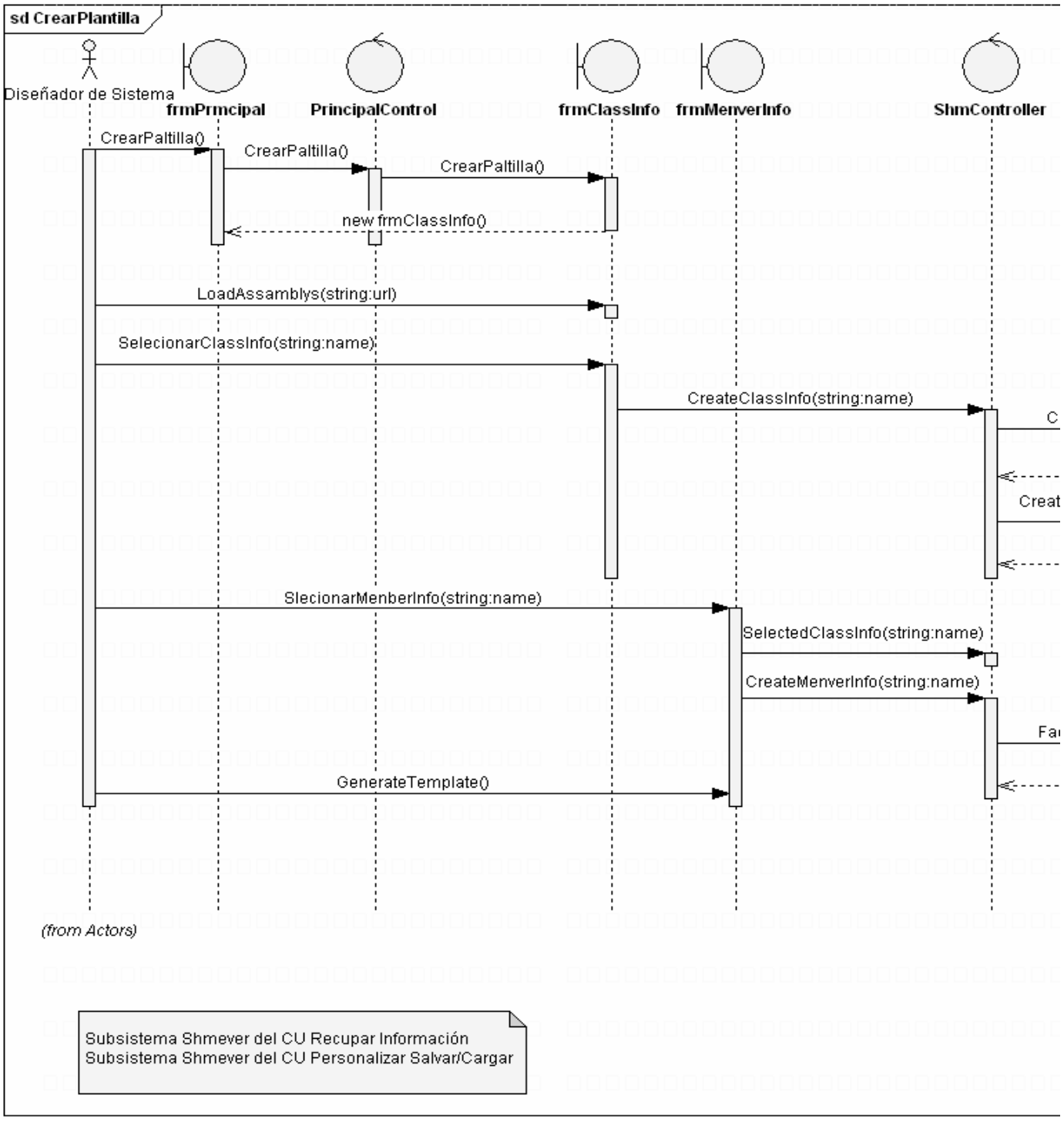
- **ControlSurrogate**

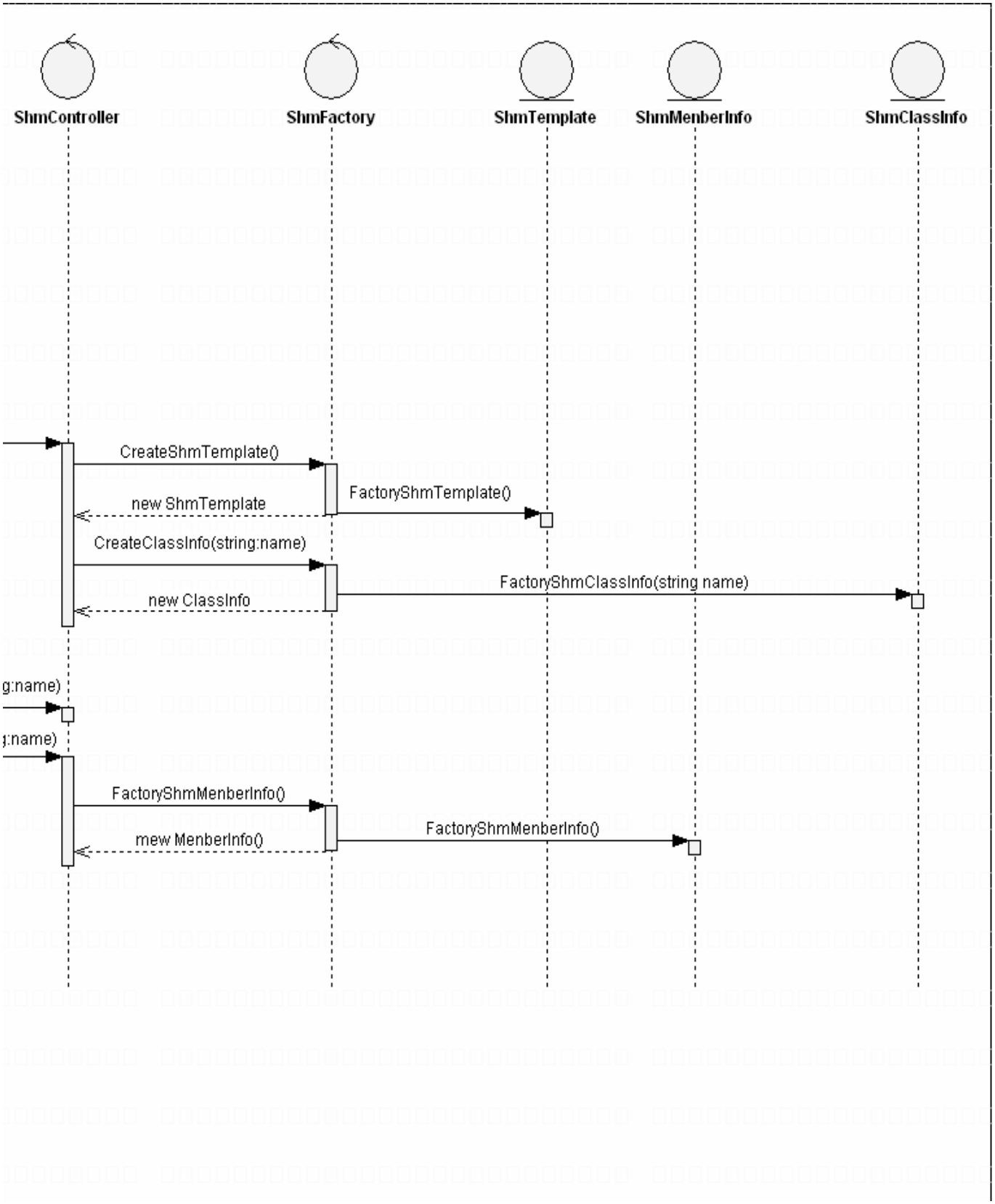
Es la clase que se encarga de dar el formato, para una salva y/o carga los archivos con control de versiones, pero de forma personalizada.

Diagrama de clases

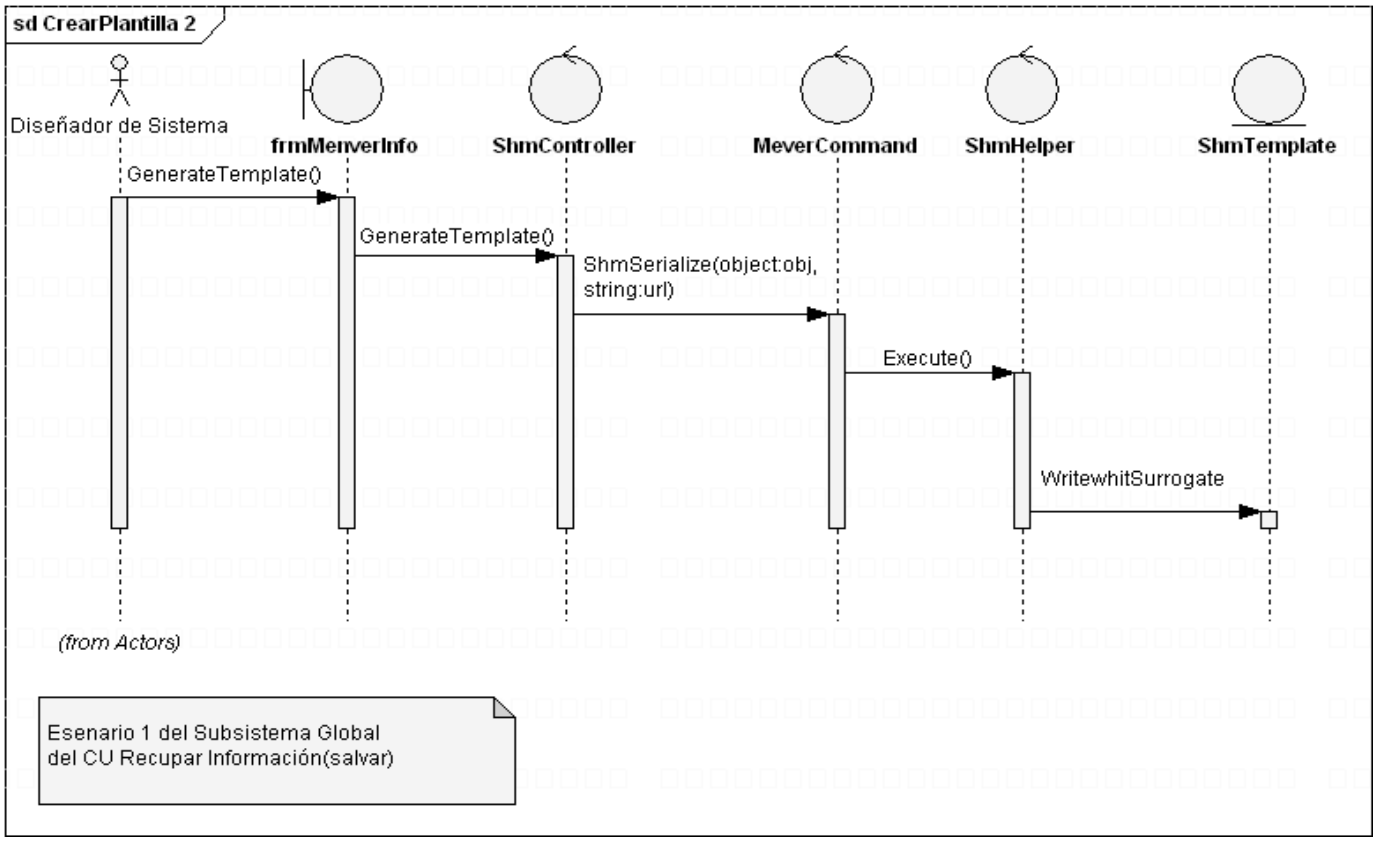


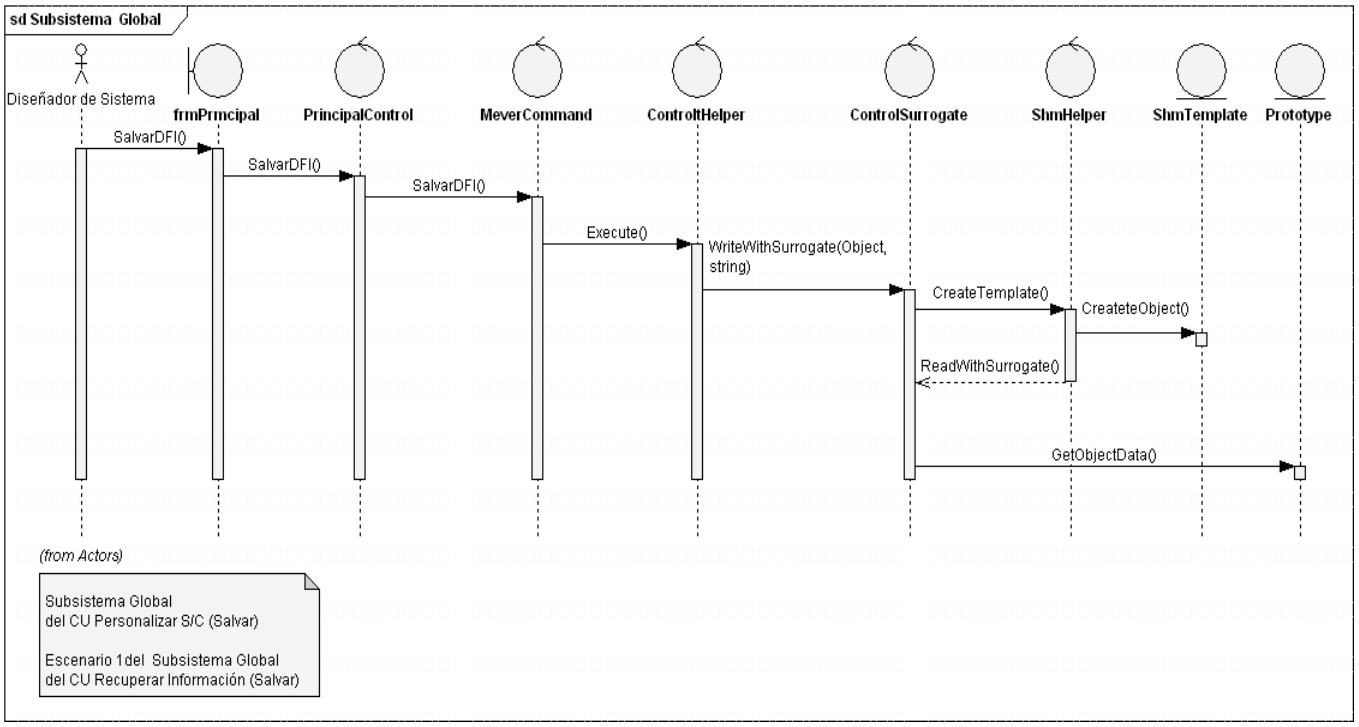
Diagramas de Secuencias

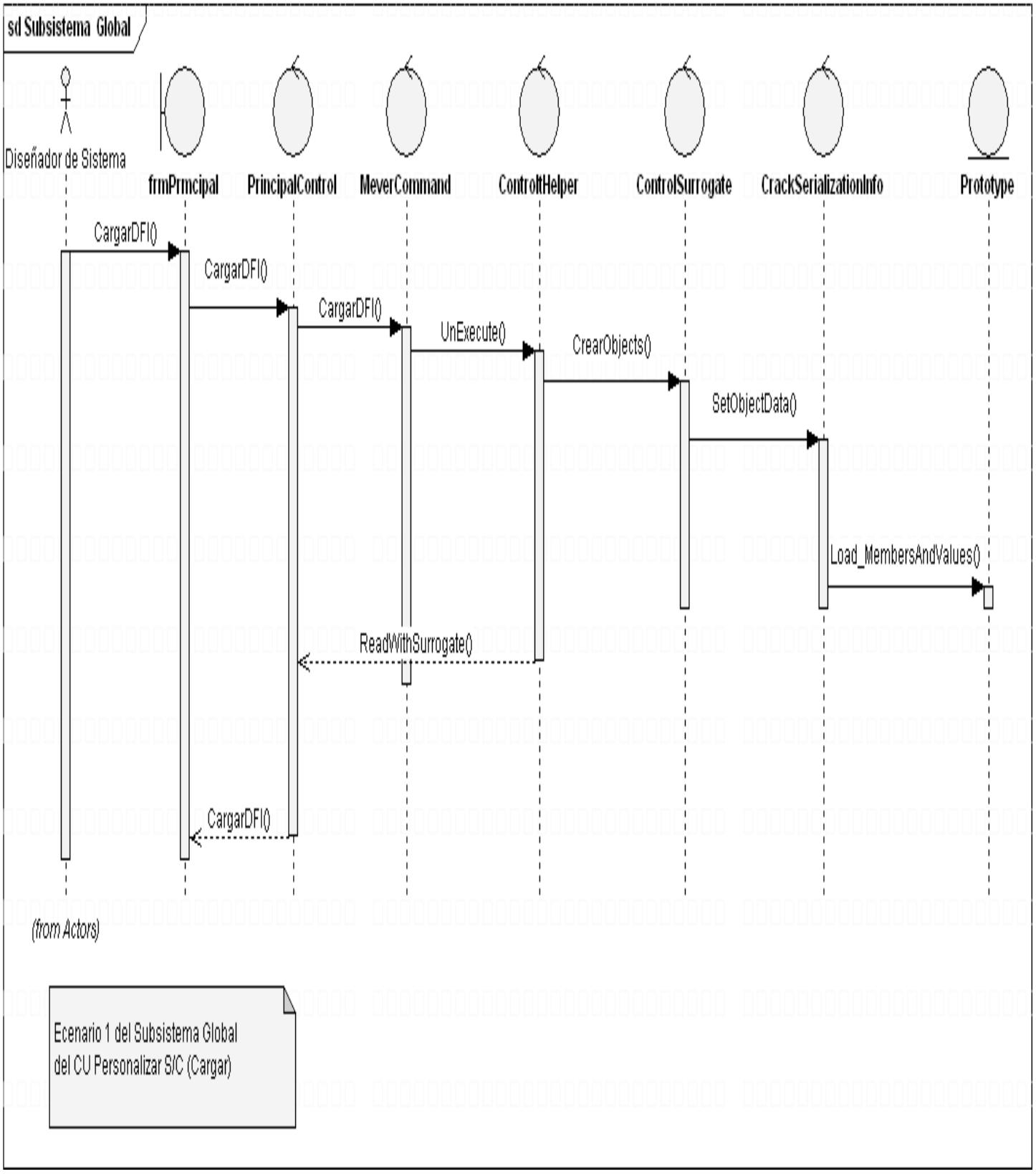



ShmController
ShmFactory
ShmTemplate
ShmMenberInfo
ShmClassInfo
CreateShmTemplate()
FactoryShmTemplate()
new ShmTemplate
CreateClassInfo(string:name)
FactoryShmClassInfo(string name)
new ClassInfo
g:name)
g:name)
FactoryShmMenberInfo()
FactoryShmMenberInfo()
mew MenberInfo()

Esta es la continuación del escenario anterior.




sd Subsistema Global
Diseñador de Sistema
frmPrincipal
PrincipalControl
MeverCommand
ControltHelper
ControlSurrogate
ShmHelper
ShmTemplate
Prototype
SalvarDFI()
SalvarDFI()
SalvarDFI()
Execute()
WriteWithSurrogate(Object, string)
CreateTemplate()
CreateteObject()
ReadWithSurrogate()
GetObjectData()
(from Actors)
Subsistema Global
del CU Personalizar S/C (Salvar)
Escenario 1del Subsistema Global
del CU Recuperar Información (Salvar)


sd Subsistema Global
Diseñador de Sistema
frmPrincipal
PrincipalControl
MeverCommand
ControlHelper
ControlSurrogate
CrackSerializationInfo
Prototype
CargarDFI()
CargarDFI()
CargarDFI()
UnExecute()
CrearObjects()
SetObjectData()
Load_MembersAndValues()
ReadWithSurrogate()
CargarDFI()
(from Actors)
Ecenario 1 del Subsistema Global
del CU Personalizar S/C (Cargar)

Caso de uso: **Recuperar Información**

- **NetHelper**

Hereda de Helper para utilizar la estrategia de salvar y/o cargar un fichero que tiene por defecto .net framework.

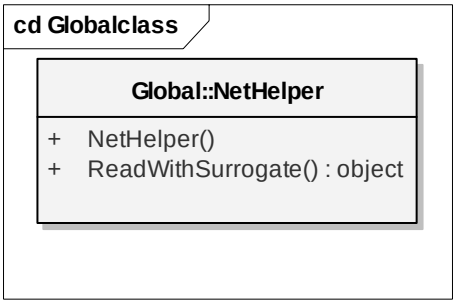
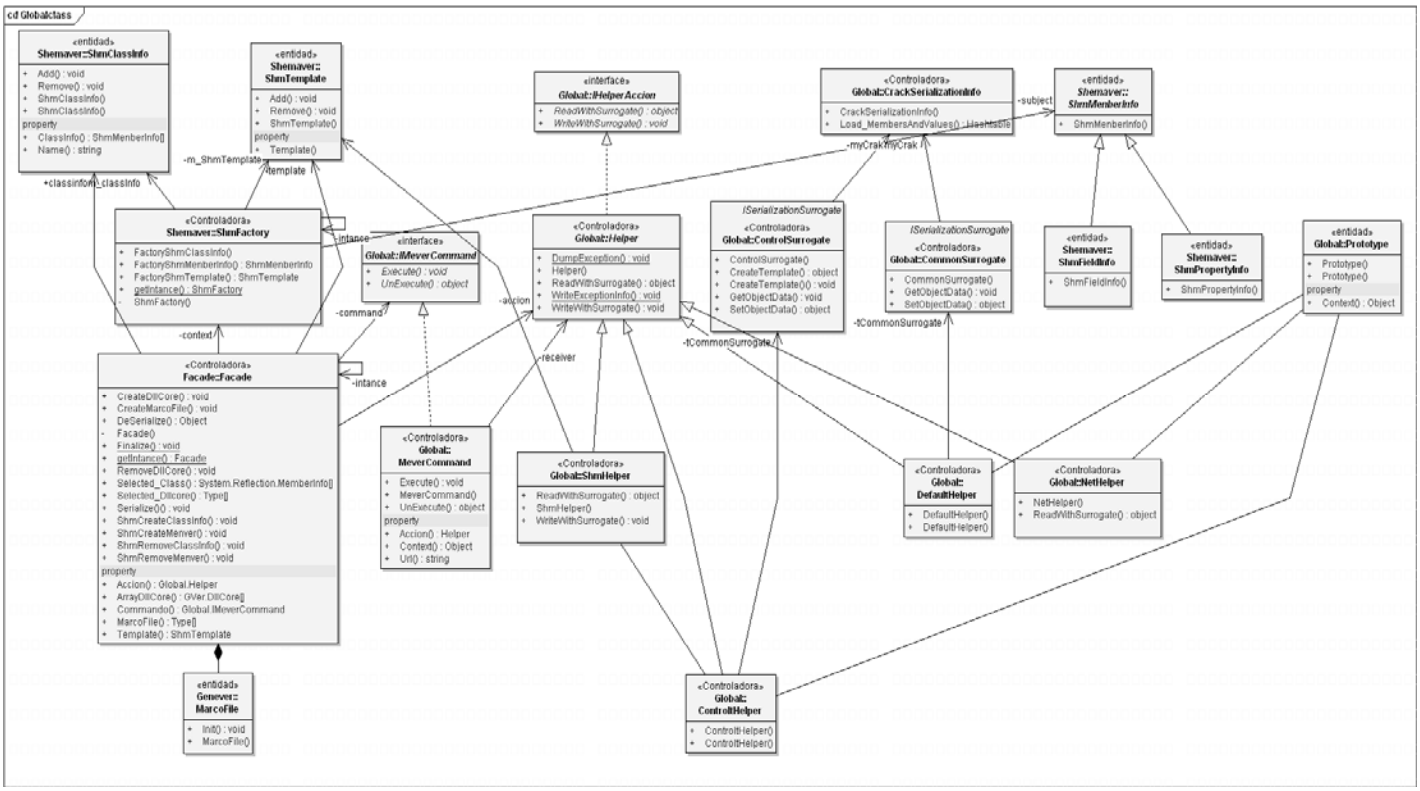
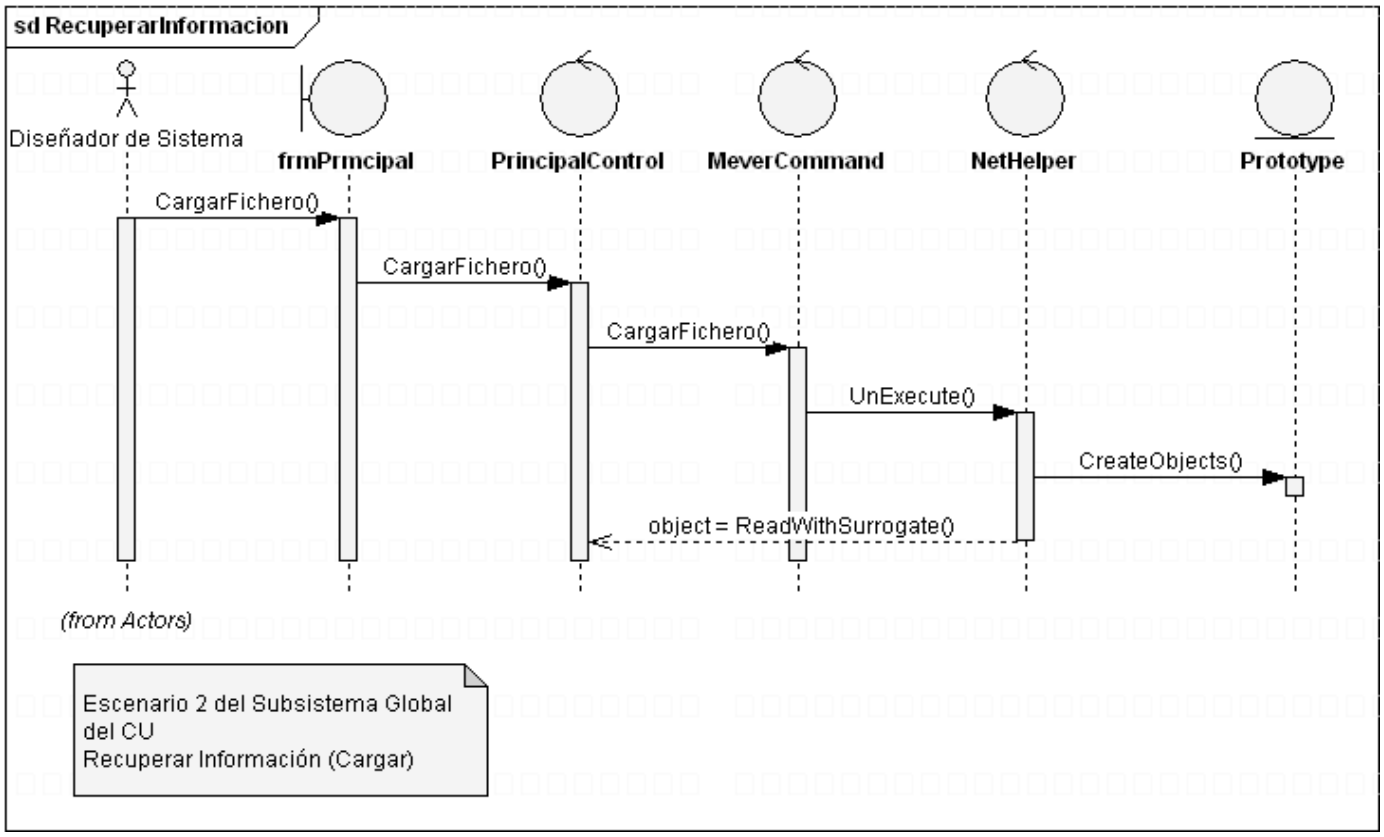
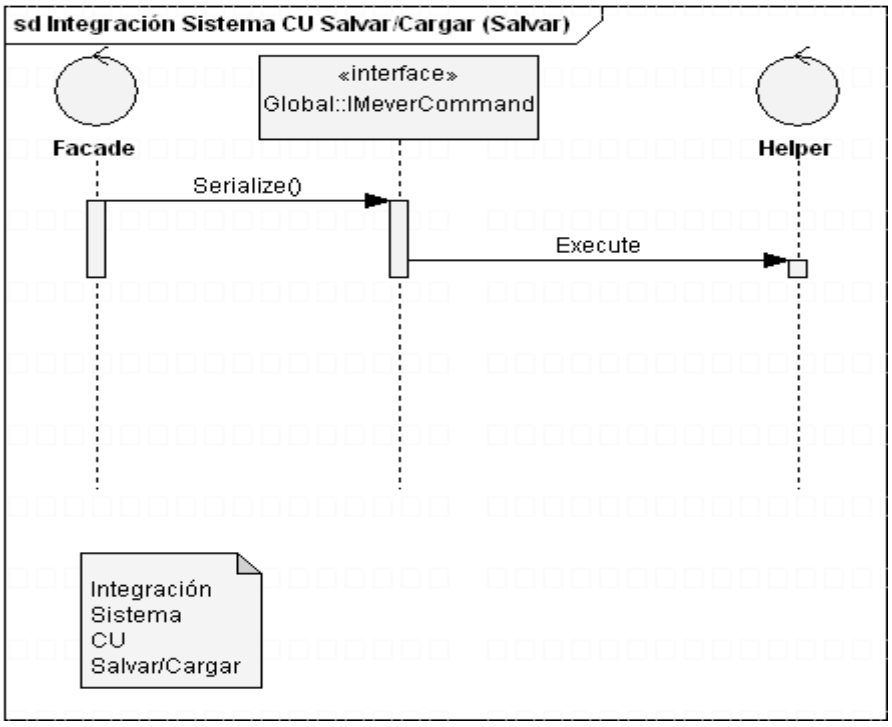


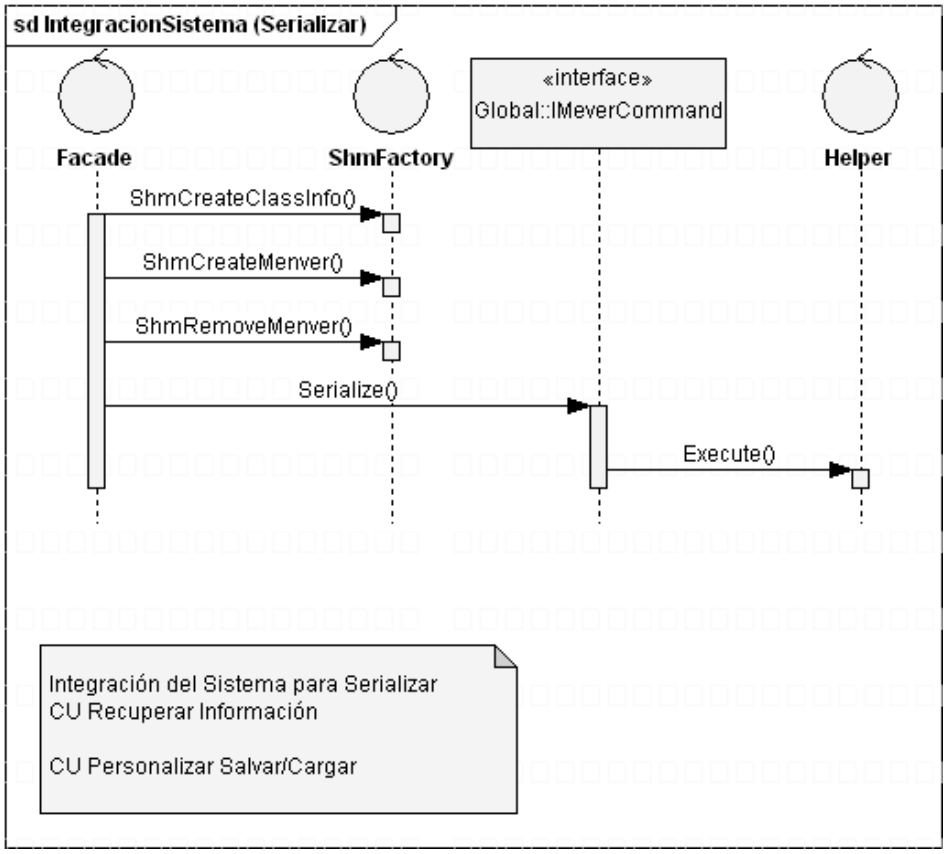
Diagrama de clases

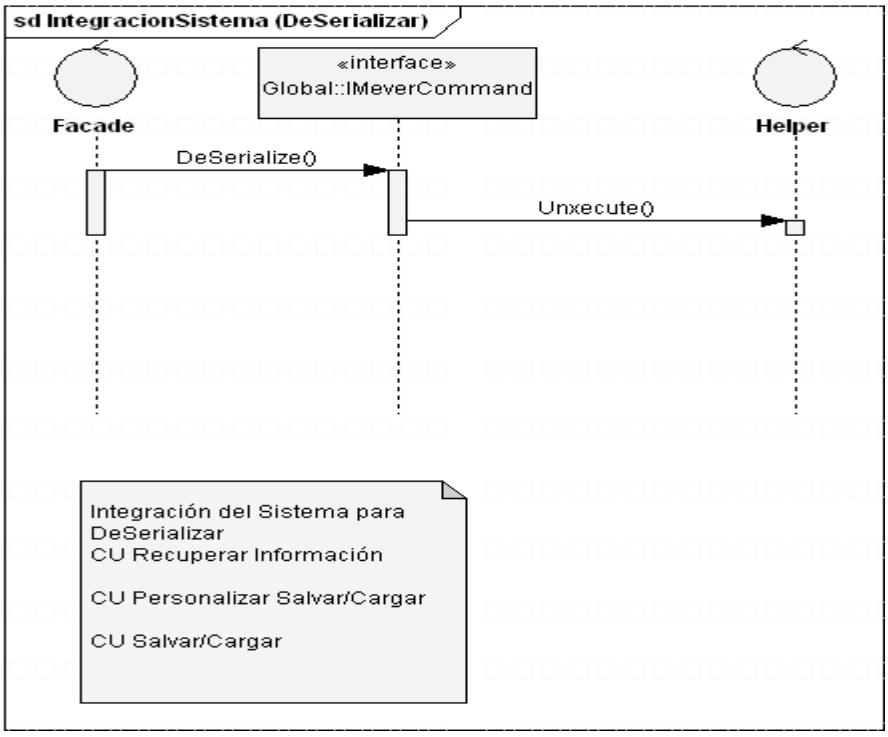


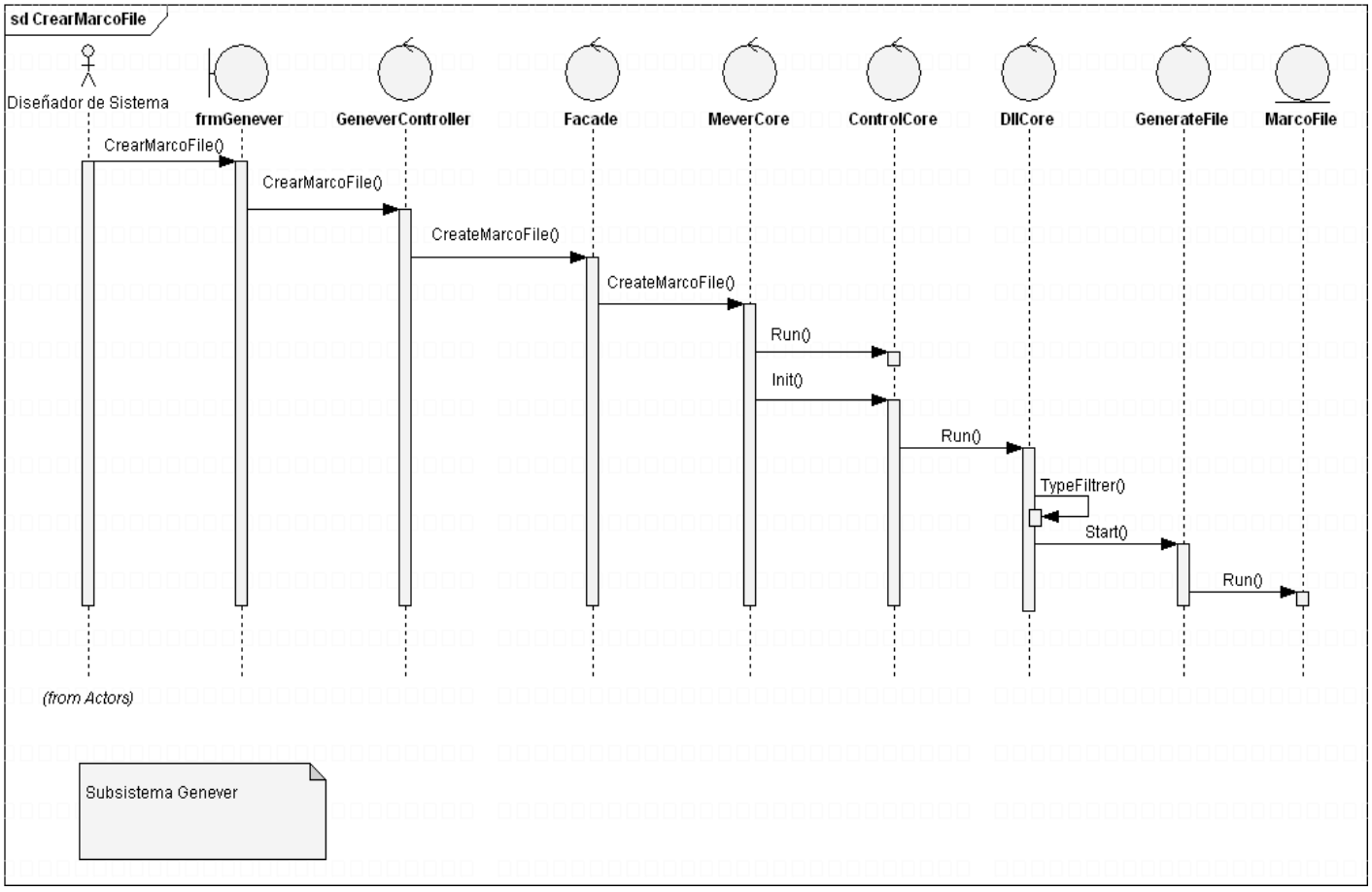
Diagramas de Secuencias



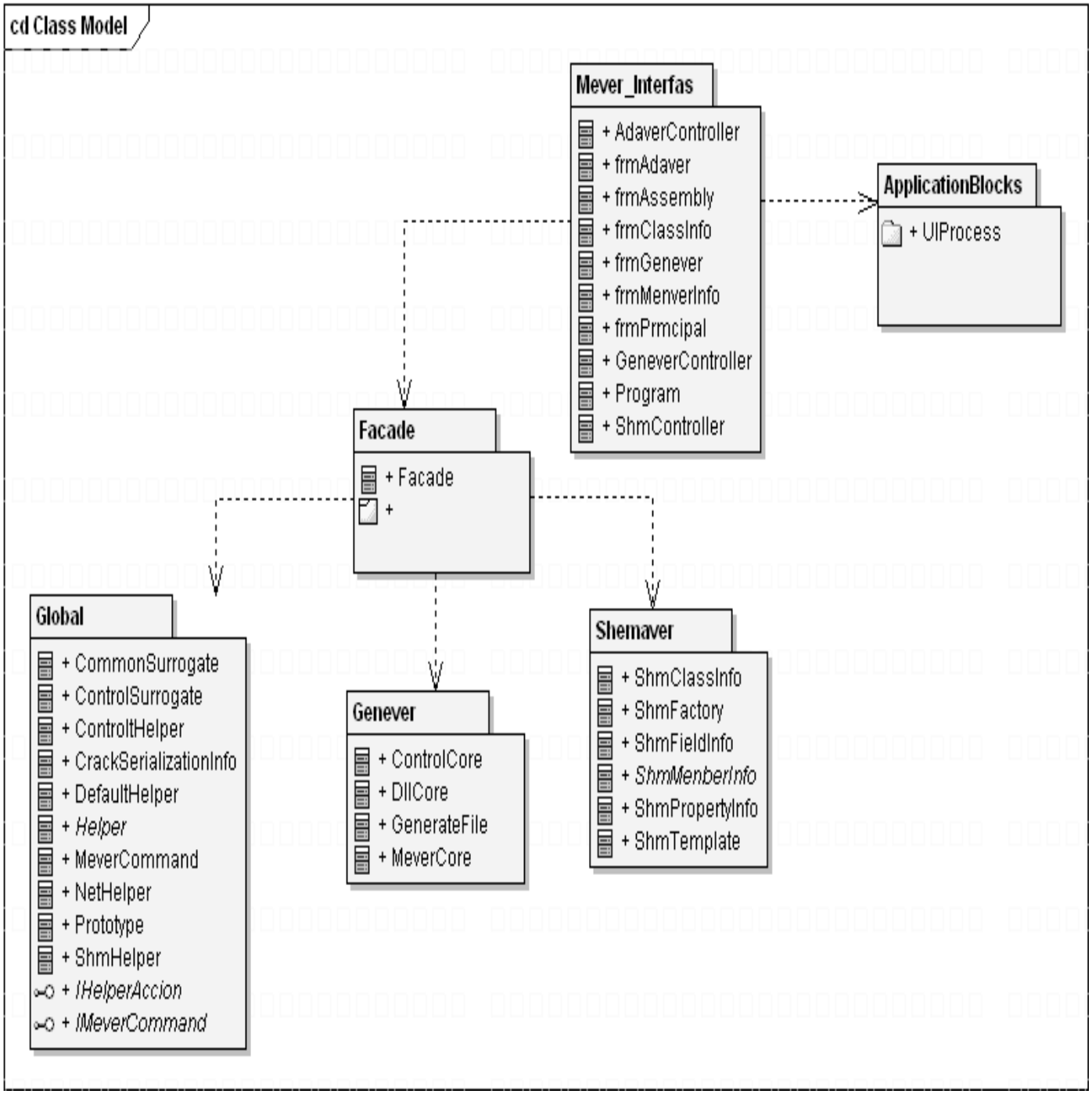

sd Integración Sistema CU Salvar/Cargar (Salvar)
«interface»
Global::IMeverCommand
Facade
Helper
Serialize()
Execute
Integración
Sistema
CU
Salvar/Cargar


sd IntegracionSistema (Serializar)
Facade
ShmFactory
«interface»
Global::IMeverCommand
Helper
ShmCreateClassInfo()
ShmCreateMenver()
ShmRemoveMenver()
Serialize()
Execute()
Integración del Sistema para Serializar
CU Recuperar Información
CU Personalizar Salvar/Cargar

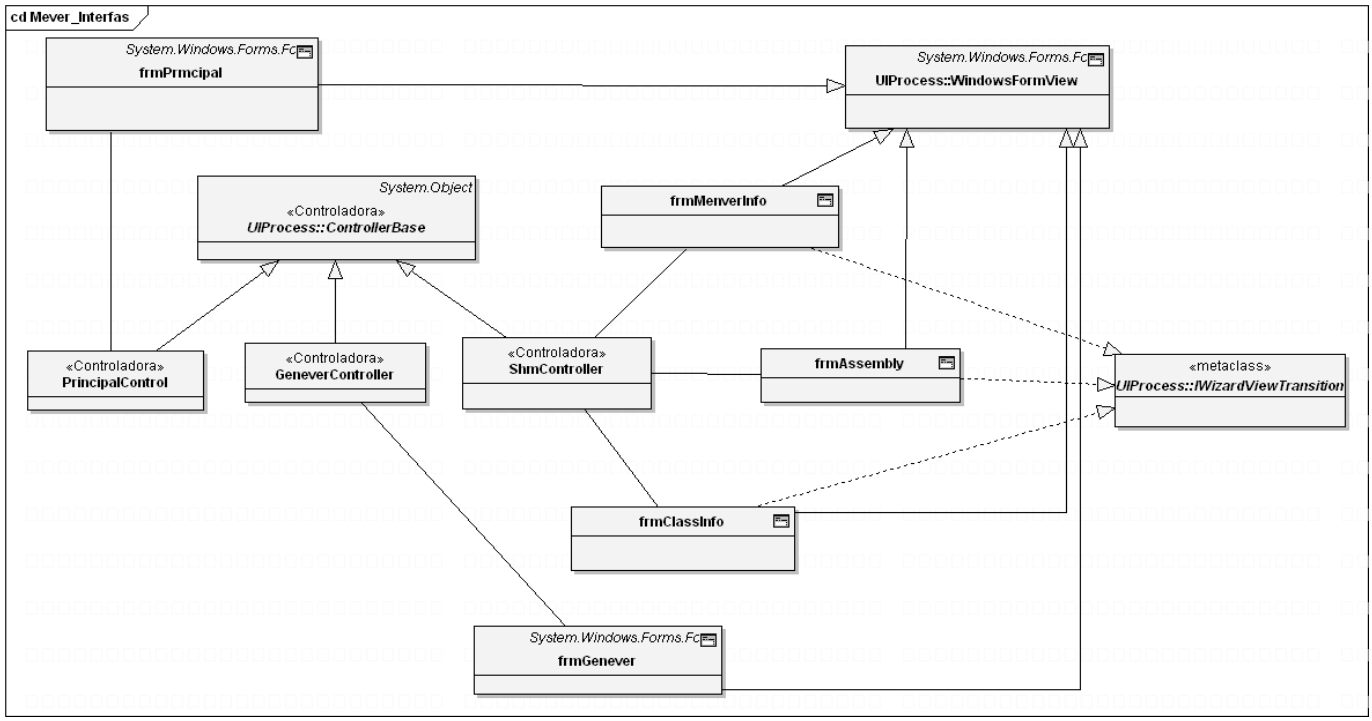

sd IntegracionSistema (DeSerializar)
«interface»
Global::IMeverCommand
Facade
Helper
DeSerialize()
Unxecute()
Integración del Sistema para DeSerializar
CU Recuperar Información
CU Personalizar Salvar/Cargar
CU Salvar/Cargar


sd CrearMarcoFile
Diseñador de Sistema
frmGenever
GeneverController
Facade
MeverCore
ControlCore
DllCore
GenerateFile
MarcoFile
CrearMarcoFile()
CrearMarcoFile()
CreateMarcoFile()
CreateMarcoFile()
Run()
Init()
Run()
TypeFiltrer()
Start()
Run()
(from Actors)
Subsistema Genever

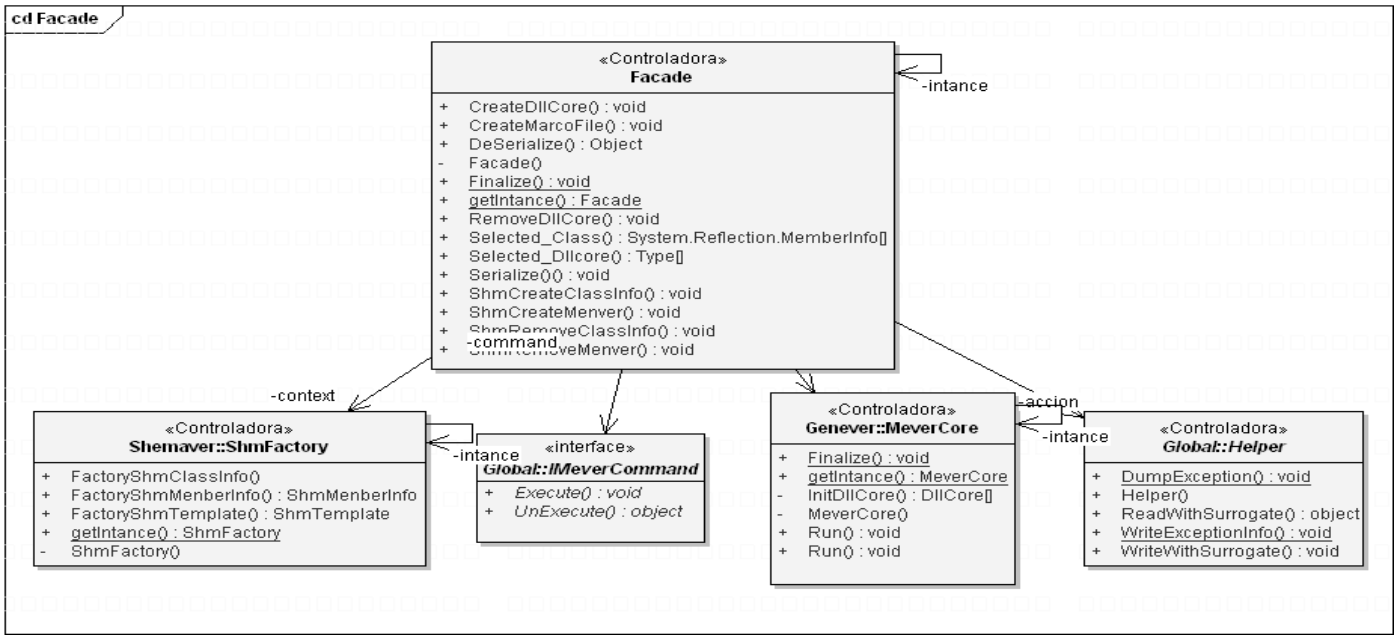
Vista modelo diseño



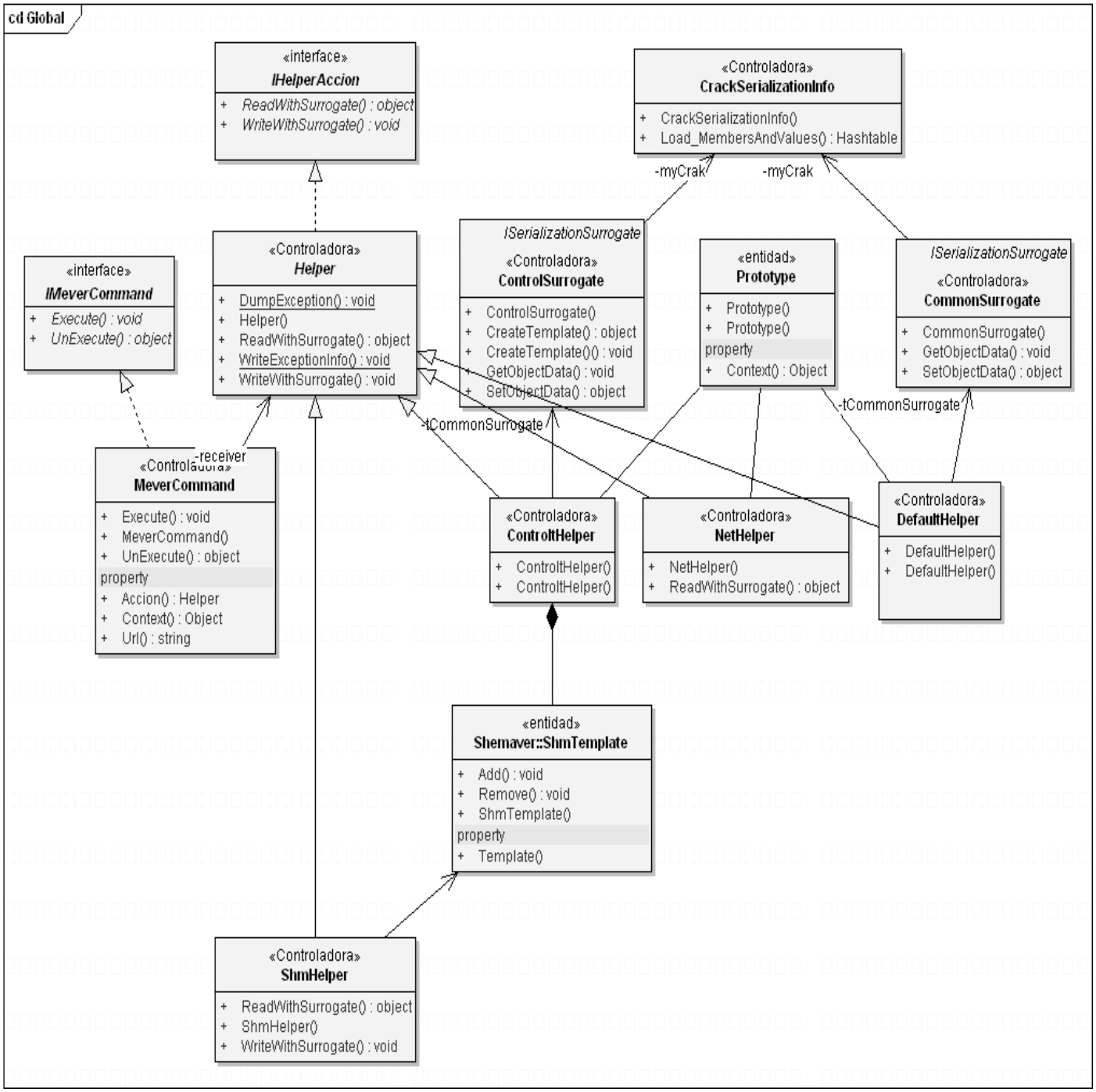
Subsistema Mever_Interfas



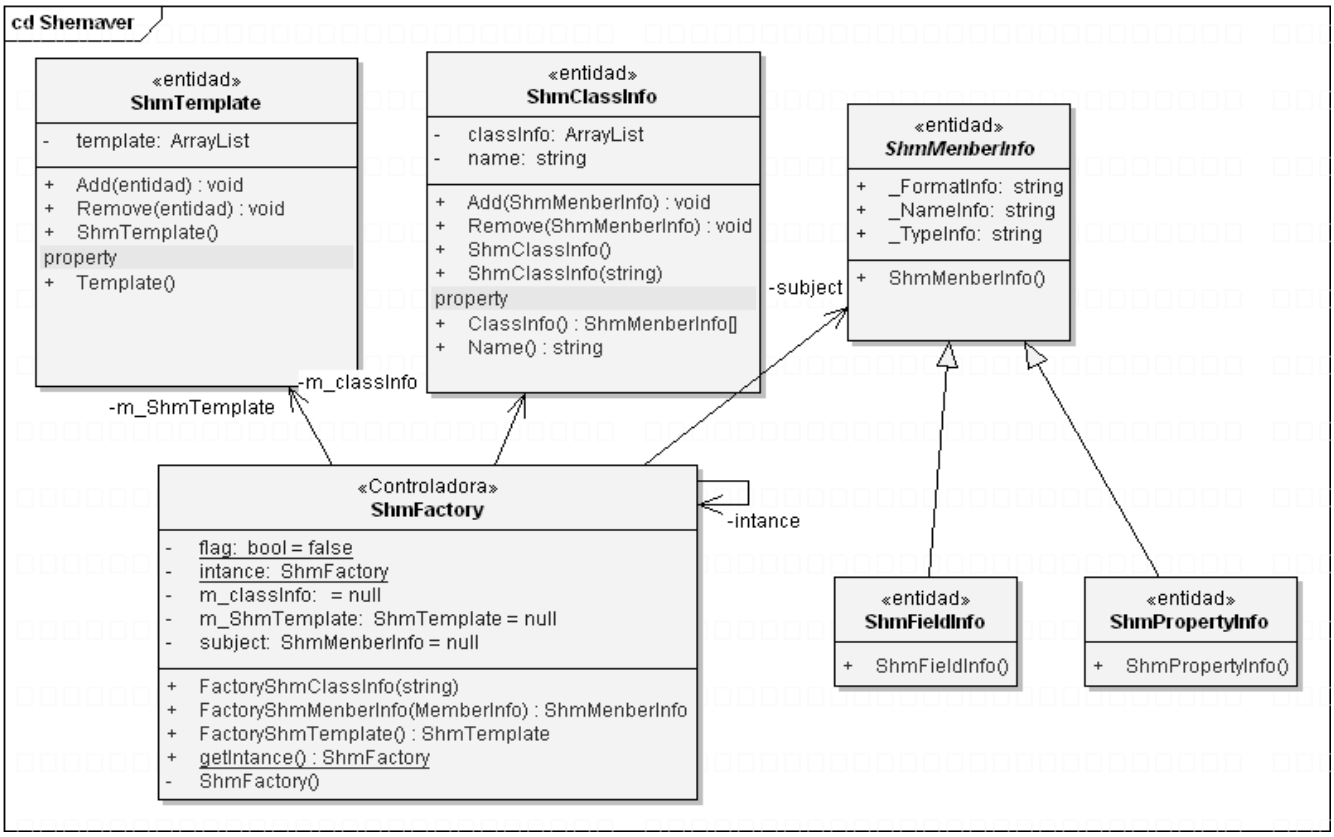
Subsistema Facade



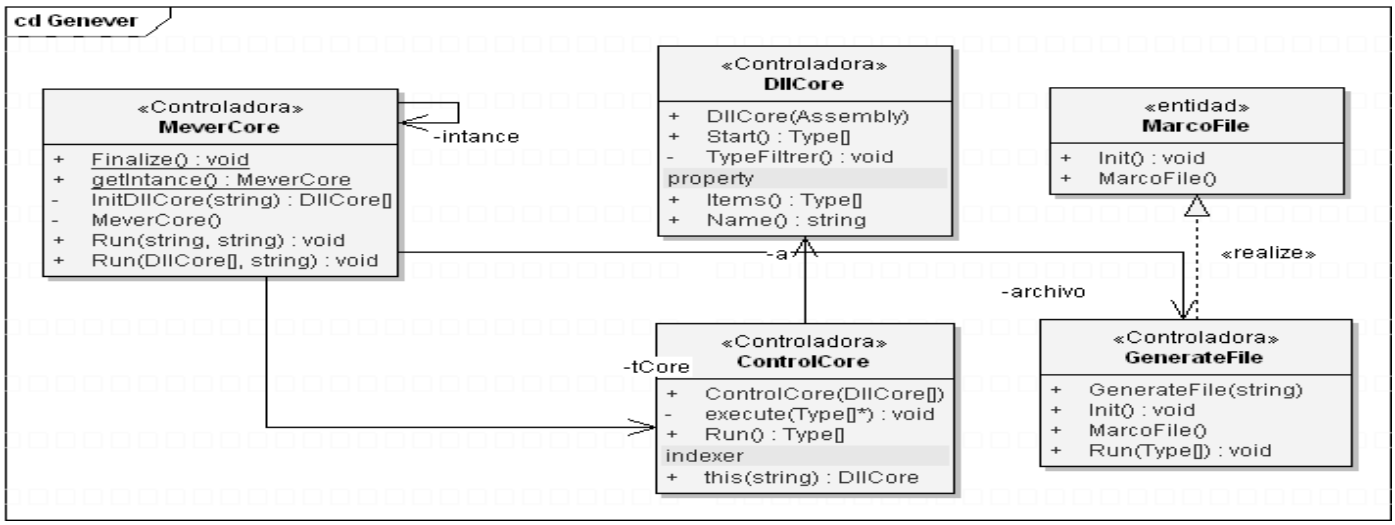
Subsistema Global



Subsistema Shemaver



Subsistema Genever



CAPITULO 3.ANALISIS DE RESULTADOS.

3.1 Introducción.

Este capítulo está dedicado a evaluar la calidad del diseño del sistema, a partir de diferentes métricas para la programación orientadas a objetos y criterios que plantean una serie de autores. También se explicará el modelo de calidad del producto de software ISO 9126 (ISO/IEC, 2001) [18], que da una definición de calidad de producto software en base a atributos. El objetivo no es necesariamente alcanzar una calidad perfecta, sino la necesaria y suficiente para cada contexto de uso a la hora de la entrega y del uso por parte de los usuarios.

Así, la calidad del software se puede medir a través de estos atributos (aunque puede haber algunos internos), que representan diferentes visiones de la misma. Los más importantes son la fiabilidad, mantenibilidad y usabilidad. El estándar IEEE Software Quality Metrics también establece una descomposición de factores de calidad. Existe otro punto de vista en la definición de calidad: es el llamado *subjetivo*, en el que no se toma un modelo de calidad prefijado, sino que para un producto dado se definen atributos de calidad conjuntamente con el usuario. Una visión más restrictiva, pero habitual, de la calidad consiste en asociarla exclusivamente con la ausencia de "defectos".

Cuando se hace mención de la calidad de un producto, según la ISO 9126 no se debe hablar de calidad en general sino de calidad interna y calidad externa afirmándose que la calidad interna influye de forma directa sobre la calidad externa del software. Una buena calidad interna asegura directamente la calidad observada en atributos del uso del producto.

3.2 Diferentes aspectos de la calidad.

- Métricas Internas ISO/IEC 9126-**:** medible a partir de las características intrínsecas, como el código fuente.
- Métricas Externas ISO/IEC 9126-2**:** medible en el comportamiento del producto, como en una prueba.
- Métricas de Calidad en el uso ISO/IEC 9126-4**:** durante la utilización efectiva por parte del usuario.

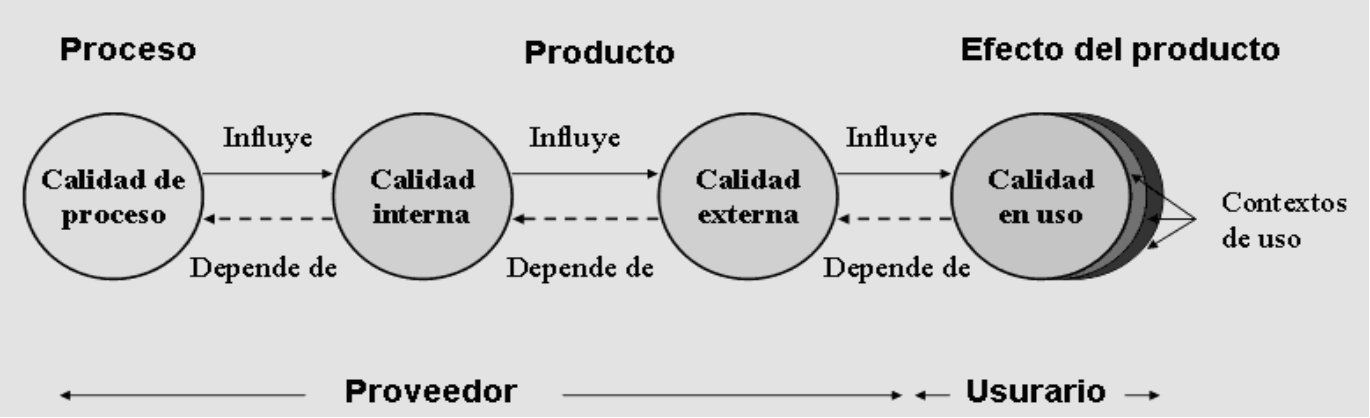


Ilustración 11.Proceso de calidad de producto software: ISO 9126.

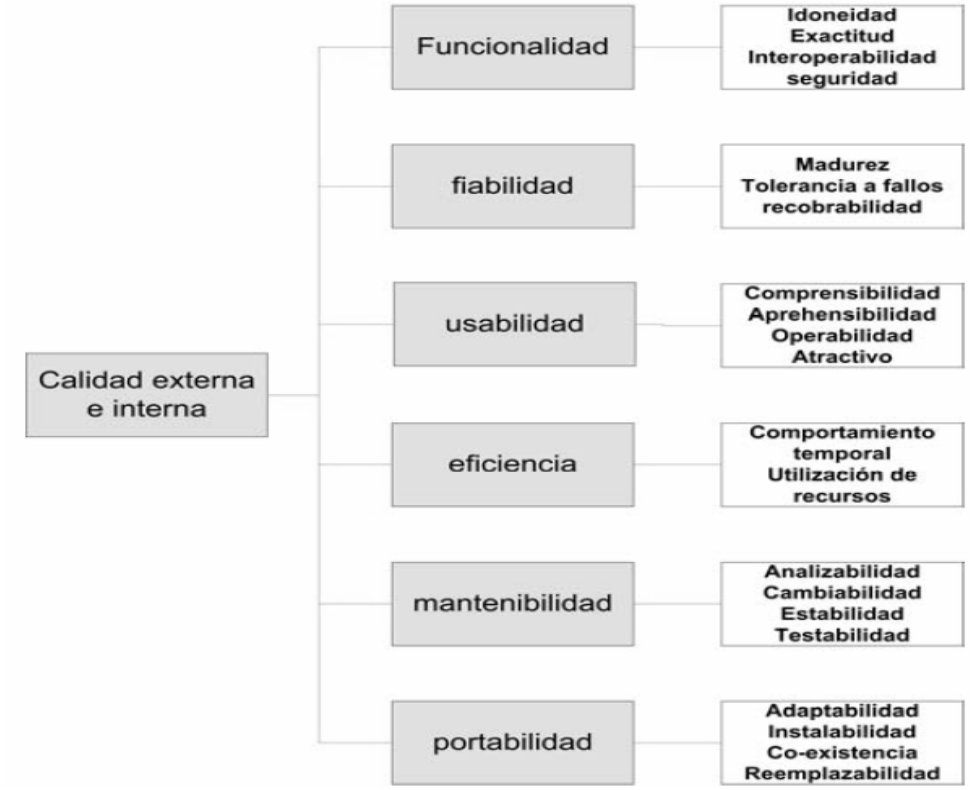


Ilustración 12. Atributos para medir la calidad de un software.

Este modelo permite especificar la calidad de producto y evaluarla desde perspectivas diferentes:

- Adquisición.
- Requisitos.
- Desarrollo.
- Uso.
- Mantenimiento.
- Aseguramiento de Calidad.

3.2.1 Atributos de la calidad.

- **Funcionalidad.**
    - **Adecuación.**

Capacidad del producto software para proporcionar un conjunto apropiado de funciones para tareas y objetivos de usuario especificados.

- **Exactitud.**

Capacidad del producto software para proporcionar los resultados o efectos correctos o acordados, con el grado necesario de precisión.

- **Interoperabilidad.**

Capacidad del producto software para interactuar con uno o más sistemas especificados.

- **Seguridad de acceso.**

Capacidad del producto software para proteger información y datos de manera que las personas o sistemas no autorizados no puedan leerlos o modificarlos, al tiempo que no se deniega el acceso a las personas o sistemas autorizados.

- **Cumplimiento funcional.**

Capacidad del producto software para adherirse a normas, convenciones o regulaciones en leyes y prescripciones similares relacionadas con funcionalidad.

- **Fiabilidad.**
    - **Madurez.**

    Capacidad del producto software para evitar fallar como resultado de fallos en el software.

    - **Tolerancia a fallos.**

    Capacidad del software para mantener un nivel especificado de prestaciones en caso de fallos software o de infringir sus interfaces especificados.

    - **Capacidad de recuperación.**

    Capacidad del producto software para reestablecer un nivel de prestaciones especificado y de recuperar los datos directamente afectados en caso de fallo.

    - **Cumplimiento de la fiabilidad.**

    Capacidad del producto software para adherirse a normas, convenciones o regulaciones relacionadas con al fiabilidad.

- **Usabilidad.**
    - **Capacidad para ser entendido.**

    Capacidad del producto software que permite al usuario entender si el software es adecuado y cómo puede ser usado para unas tareas o condiciones de uso particulares.

    - **Capacidad para ser aprendido.**

    Capacidad del producto software que permite al usuario aprender sobre su aplicación.

- **Capacidad para ser operado.**

Capacidad del producto software que permite al usuario operarlo y controlarlo.

- **Capacidad de atracción.**

Capacidad del producto software para ser atractivo al usuario.

- **Cumplimiento de la usabilidad.**

Capacidad del producto software para adherirse a normas, convenciones, guías de estilo o regulaciones relacionadas con la usabilidad.

- **Eficiencia.**
  - **Comportamiento temporal.**

Capacidad del producto software para proporcionar tiempos de respuesta, tiempos de proceso y potencia apropiados, bajo condiciones determinadas.

  - **Utilización de recursos.**

Capacidad del producto software para usar las cantidades y tipos de recursos adecuados cuando el software lleva a cabo su función bajo condiciones determinadas.

  - **Cumplimiento de la eficiencia.**

Capacidad del producto software para adherirse a normas o convenciones relacionadas con la eficiencia.

- **Mantenibilidad.**
    - **Capacidad para ser analizado.**

    Es la capacidad del producto software para serle diagnosticadas deficiencias o causas de los fallos en el software, o para identificar las partes que han de ser modificadas.

    - **Capacidad para ser cambiado.**

    Capacidad del producto software que permite que una determinada modificación sea implementada.

    - **Estabilidad.**

    Capacidad del producto software para evitar efectos inesperados debidos a modificaciones del software.

    - **Capacidad para ser probado.**

    Capacidad del producto software que permite que el software modificado sea validado.

    - **Cumplimiento de la mantenibilidad.**

    Capacidad del producto software para adherirse a normas o convenciones relacionadas con la mantenibilidad.

- **Portabilidad.**
    - **Adaptabilidad.**

    Capacidad del producto software para ser adaptado a diferentes entornos especificados, sin aplicar acciones o mecanismos distintos de aquellos proporcionados para este propósito por el propio software considerado.

    - **Instalabilidad.**

    Capacidad del producto software para ser instalado en un entorno especificado.

    - **Coexistencia.**

    Capacidad del producto software para coexistir con otro software independiente, en un entorno común, compartiendo recursos comunes.

    - **Capacidad para reemplazar.**

    Capacidad del producto software para ser usado en lugar de otro producto software, para el mismo propósito, en el mismo entorno.

    - **Cumplimiento de la portabilidad.**

    Capacidad del producto software para adherirse a normas o convenciones relacionadas con la portabilidad.

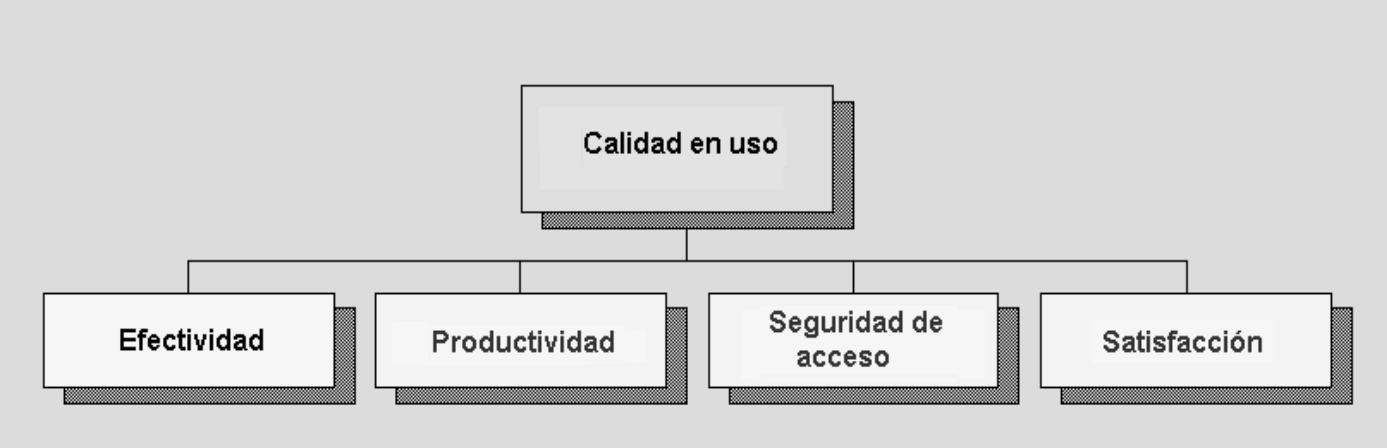


Ilustración 13. Modelo de calidad para calidad en uso.

3.2.2 Atributos del modelo de calidad para calidad en uso.

- **Efectividad.**

Capacidad del producto software para permitir a los usuarios alcanzar objetivos especificados con exactitud y completitud, en un contexto de uso especificado.

- **Productividad.**

Capacidad del producto software para permitir a los usuarios gastar una cantidad adecuada de recursos con relación a la efectividad alcanzada, en un contexto de uso especificado.

- **Seguridad física.**

Capacidad del producto software para alcanzar niveles aceptables del riesgo de hacer daño a personas, al negocio, al software, a las propiedades o al medio ambiente en un contexto de uso especificado.

- **Satisfacción.**

Capacidad del producto software para satisfacer a los usuarios en un contexto de uso especificado.

3.3 Métricas para la evaluación del diseño.

3.3.1 Métricas OO.

En el caso particular de la programación orientada a objetos, son muchos los conjuntos de métricas OO publicados que utilizan propiedades como, cohesión, acoplamiento, encapsulamiento, complejidad y herencia para evaluar la calidad del diseño, que, sin ser todas específicas del diseño OO, son lo suficientemente concretas y además se pueden relacionar matemáticamente con los atributos generales de diseño.

3.3.1.1 Métrica de Chindamber y Kemerer.

- **WMC** [19]: Los autores sugieren que WMC es una medida de la complejidad de una clase. Clases con un gran número de métodos que requieren más tiempo y esfuerzo para desarrollarlas y mantenerlas, ya que influirán en las subclases heredando todos sus métodos. Además, estas clases tienden a ser específicas de la aplicación, con lo que se limita su posibilidad de reutilización. Lorenz y Kidd plantean un umbral de 40 o 20, dependiendo de si las clases son o no de interfaz de usuario.

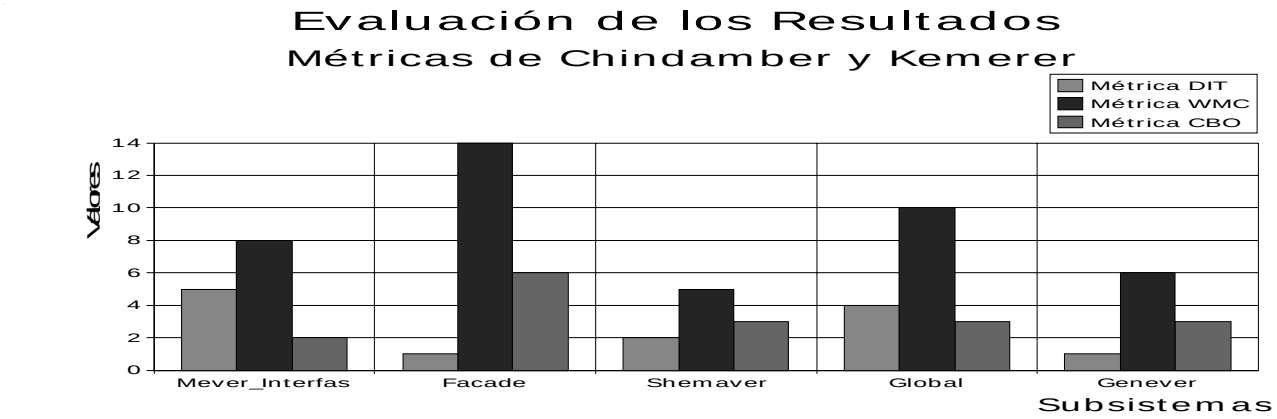
3.3.1.2 Métrica para la herencia.

- **DIT** [19]: En general la herencia se utiliza poco. Distintos autores han encontrado una correlación positiva entre DIT y el número de problemas emitidos por el usuario, poniendo en duda el uso efectivo de la herencia. Por su parte, otros autores sugieren un umbral de 6 niveles como indicador de un abuso en la herencia en distintos lenguajes de programación (C++, Smalltalk).

3.3.1.3 Métricas que se consideran a nivel de acoplamiento.

- **CBO**: Coupling Between Objects (acoplamiento entre objetos) de una clase se define como el número de clases a las cuales una clase está ligada. Se da dependencia entre dos clases cuando una de ellas usa métodos o variables de la otra clase. Las clases relacionadas por herencia no se tienen en cuenta. Los autores proponen que esta métrica sea un indicador del esfuerzo necesario para el mantenimiento y las pruebas.

El siguiente gráfico hace un reporte del comportamiento que tienen los diferentes subsistemas a partir de las métricas mencionadas:



Se puede observar que el sistema tiene un nivel de herencia intermedio, según la métrica DIT, lo que posibilita una reusabilidad potencial, ya que aprovecha los mecanismos de herencia proporcionado por la programación orientada a objetos, también facilita que los objetos no lleguen a ser demasiados complejos lo que facilita que estos sean fáciles de probar y reutilizar, sin caer en los compromisos que lleva la herencia.

Estas métricas (DIT) presenta medidas objetivas para la complejidad de las clases y ha sido validada teóricamente por los autores al corroborar que satisfacen los axiomas de Weyuker.

De acuerdo a la métrica (WMC) el sistema tiene un nivel de complejidad intermedio, lo que facilita el mantenimiento de estas, ya que de no ser así los efectos negativos influirán directamente en las clases hijas de acuerdo a los compromisos que trae consigo la herencia, y todo esto nos lleva a la necesidad de requerir de muchos más tiempo.

Unos de los atributos que un sistema debe tener maximizados es el acoplamiento en su nivel más bajo para alcanzar una baja complejidad y una mejor modularidad. Todo esto trae consigo una mejor reutilización de las clases, una mejor comprensión de ellas y un menor esfuerzo a la hora de realizar las pruebas. Si se reduce el acoplamiento se promueve la encapsulación.

También se hizo uso distintos patrones, ya que un buen diseño orientado a objetos lleva implícito patrones [12]. Estos recursos para la elaboración de un buen diseño se han aplicado en este trabajo de la mejor forma posible y cuando se ha estimado necesario, para tratar de maximizar las propiedades de un diseño OO. Se hizo uso de los patrones GRASP [12] y los patrones de diseño GOF [1].

Estos patrones han influido positivamente pues el diseño quedó con una mejor modularidad, un bajo acoplamiento, una mejor cohesión y una mejor asignación de responsabilidades.

Actualmente existen trabajos acerca de este tema, que han establecido métricas específicas basadas en el diseño, como por ejemplo (Genero et al., 2001; Kiewkanya et al., 2004) que establecen un conjunto de métricas sobre diagramas UML (de clases y secuencias). Sin embargo pocos establecen un método completo de evaluación de calidad del diseño (los dos anteriores, por ejemplo, se centran exclusivamente en la mantenibilidad).

El framework como producto final, aunque todavía está en prueba tiene grandes expectativas por las facilidades que tiene y por alcanzar los objetivos. Si se hace una comparación detallada con otros framework podemos observar a simple vista algunas características que lo destacan.

Según las investigaciones y búsquedas en Internet, hasta ahora el único producto en Internet de este tipo es **Compactor.** Una herramienta propietaria cuyo costo llega a alcanzar los 1599 dólares.

CONCLUSIONES

Una vez finalizado este trabajo de diploma, se arribó a las siguientes conclusiones:

En el ámbito mundial existe el software Compactor que gestiona los procesos de salvar y cargar, pero mantiene la dependencia entre estos procesos y el desarrollador, es decir, el desarrollador tiene que conocer que información se necesita guardar para inyectarle código que gestione estos procesos. Además no posee un proceso de serialización personalizada.

La puesta en práctica de este framework garantiza el ahorro de tiempo de trabajo al evitar que no se confeccionen nuevamente los DFI en la nueva versión, hace posible que se salve y cargue un DFI en cualquier versión de CheSys sin perder información, logrando así satisfacer las necesidades del cliente y una total independencia de este módulo con el resto del sistema.

También se logró guardar información en un formato que posibilite seguir trabajando sobre ella, al igual posibilitó guardar información en un formato personalizado y se logró alcanzar una arquitectura escalable.

Con la culminación de este trabajo realizado se cumple con los objetivos trazados: *Desarrollar un módulo informático que permita delegar a un solo usuario el proceso de salvar/cargar y que permita cargar DFI desarrollados en versiones anteriores.*

BIBLIOGRAFIA

[1]. “Design Patterns, *Data&Object Factory.”*

Disponible en <http://www.dofactory.com/Patterns/Patterns.aspx >

[2]. “Enterprise Architect.“

Disponible en <http://www.sparxsystems.com.au/products/ea.html>

[3]. S. Pressman, Roger. Ingeniería del Software. Un enfoque práctico. McGraw Hill, 1997.

[4]. “Rup”

Disponible en <www.rational.com/products/rup/index.jsp>

[5]. “Dr. Marteens.Reflexión”

Disponible en <http://www.marteens.com/trick56.htm>

[6]. “serialización XML”

Disponible en <http://msdn2.microsoft.com/es-es/library/58a18dwa(VS.80).aspx>

[7]. “Ejemplos de serialización XML”

Disponible en <http://msdn2.microsoft.com/es- s/library/182eeyhh(VS.80).aspx>

[8]. “Object Serialization using C#”

Disponible en <http://www.codeproject.com/csharp/objserial.asp>

[9]. “XMLSerializer in .NET. *TopXML.*”

Disponible en <http://www.topxml.com>

[10]. “Diseño”

Disponible en <http://readyset.tigris.org/nonav/es/templates/design.html>

[11]. “Fowler, Martín. UML Gota a Gota. Primera Edición. Addison Wesley Longman. 1999.”

[12]. “Larman, Craig. UML y Patrones, Introducción al análisis y diseño orientado a objetos. Prentice-Hall, 2002.”

[13]. “Ceria, Santiago. Ingeniería de Software I. Casos de Uso. Un Método Práctico para Explorar Requerimientos.”

[14]. “User Interface Process Application Block(2).”
Disponible en <http://www.mildiez.net/archivos/2005/10/05/user-interface-process-application-block-y-2>
[15]. “Universidad .NET.”
Disponible en <http://www.microsoft.com/spanish/msdn/comunidad/uni.net/ (4/4/2004)>
[16]. “Curso práctico de desarrollo de aplicaciones con Visual Studio .NET.”
Disponible en <http://www.microsoft.com/spanish/msdn/comunidad/uni.net/ (4/4/2004)>
[17]. “User Interface Process Application Block (1)”
Disponible en <http://www.mildiez.net/archivos/2005/09/24/user-interface-process-application-block-1/>
[18]. “OLMEDILLA ARREGUI, JUAN JOSÉ: REVISIÓN SISTEMÁTICA DE MÉTRICAS DE DISEÑO ORIENTADO A OBJETOS.”
[19]. “Salamanca Escorial, Jorge. Métricas para modelos conceptuales. Modelos conceptuales Orientados a Objetos.”

GLOSARIO DE TERMINOS

DFI: Diagrama de Flujo de Información, se refiere a los diagramas que dibujan los ingenieros químicos u otros especialistas que modelan la estructura de la fábrica que se desea simular.

Framework: Marco de trabajo o Microarquitectura que proporciona una plantilla incompleta para sistemas de un determinado dominio, puede tratarse por ejemplo un subsistema distribuido para ser ampliado y reutilizado.

LINUX: Sistema operativo portable, flexible, potente, con entorno programable, multiusuario y multitarea, muy difundido.

Mever: Sistema mergedor de Versiones.

Release: Se refiere a un producto final, preparado para lanzarse como versión definitiva a menos que aparezcan errores que lo impidan.

XML: Extensible Markup Language (Lenguaje extensible de etiquetas). Es un meta-lenguaje que nos permite definir lenguajes de marcado adecuado a usos determinados. Se propone como lenguaje de bajo nivel (a nivel de aplicación, no de programación) para intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo, y casi cualquier cosa que podamos pensar.

OPINIÓN DEL TUTOR DEL TRABAJO DE DIPLOMA

Título: **Módulo Salvar-Cargar del simulador para la Industria Química desde el rol de diseñador de sistema.**
Autores: **Enaide Corrales Fernández.**
Tutor: **Ing. Arnaldo Gandol Álvarez.**

El diplomante mostró gran independencia en el desarrollo del trabajo de diploma, tomando en cuenta además que su tutor no se encontró de forma presencial en todo el curso del trabajo, además de una alta creatividad a lo largo de todo el desarrollo. En la modelación previa al producto obtenido, el estudiante incorporó técnicas clásicas y robustas como son los patrones de diseño, separación en capas y algunas otras buenas prácticas de programación.

El trabajo fue realizado consultando documentación actualizada y una responsabilidad enorme del diplomante, pues este tema de serialización no es muy conocido, el mismo presenta buena ortografía, redacción y concordancia entre las ideas, además de una buena estructuración de los contenidos y un alto rigor científico.

Por todo lo anteriormente expresado considero que el estudiante está apto para ejercer como Ingeniero Informático; y propongo que se le otorgue al Trabajo de Diploma la calificación de 5 puntos. Considero además que el mismo cuenta con la calidad y rigor científico necesarios para ser presentado en eventos y publicaciones.

Ing. Arnaldo Gandol Álvarez.

_____ _____
Firma Fecha